HIGHER ORDER SOFTWARE, INC.
843 Massachusetts Avenue
Cambridge, MA 02139

LEVEL II

TECHNICAL REPORT #12

THE APPLICATION OF HOS TO PLRS

November 1977

Prepared for

U.S. Army Electronics Command
Ft. Monmouth, New Jersey

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Final Report TR12 ✓	2. GOVT ACCESSION NO.: AD-A075990
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE *(and Subtitle)*: The Application of HOS to PLRS,	5. ~~TYPE OF REPORT & PERIOD COVERED~~: Final Report for Period 27 May-31 Oct 1977
	6. PERFORMING ORG. REPORT NUMBER: Technical Report #12
7. AUTHOR(s): Higher Order Software, Inc.	8. CONTRACT OR GRANT NUMBER(s): DAAB07-77-C-3049
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Higher Order Software, Inc. ✓ 843 Massachusetts Avenue Cambridge, MA 02139 (as of 12/9/77, 806 Massachusetts Ave. Cambridge, MA 02139)	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PLRS
11. CONTROLLING OFFICE NAME AND ADDRESS: Electronics Systems Procurement Branch Procurement and Production Directorate U.S. Army Electronics Command, Ft. Monmouth, NJ	12. REPORT DATE: Nov 1977
	13. NUMBER OF PAGES: 142
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: [illegible]	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Position Location Reporting System (PLRS), communications, methodology, specification, design, verification, axiomatic.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The purpose of the PLRS project was to demonstrate the advantages of applying an effective methodology to a large system development process. A portion of the most complex module (the Network Manager) of the PLRS system was selected and that module was specified in terms of Higher Order Software (HOS), using the specification language, AXES, with graphical representation in terms of control maps. This module includes data type and control structure definitions, which were specifically defined to be used in common with the PLRS system environment. We discuss, here, how such a technique was able to (1) accelerate

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

the learning process of PLRS; (2) accelerate the specification process in the redefinition of a PLRS module; (3) serve as a verification and validation aid for the specification of PLRS; (4) aid in establishing overall design goals; (5) potentially enhance existing techniques of an ongoing development; and (6) provide management visibility with respect to the overall PLRS system. As a result of our finding on this effort, we discuss recommendations and their implications both for PLRS and future efforts.

ACKNOWLEDGEMENTS

This report was prepared under Contract No. DAAB07-77-C-3049 by the U.S. Army Electronics Command, Ft. Monmouth, New Jersey.

With respect to the staff of Higher Order Software, Inc., we would like to thank, in particular, Craig Thiersch for major technical contributions to this effort, which include Sections 2,3, and 4 of this report. In addition, we would like to thank Charles Musselman for Section 5, Steve Kenton for work performed on this project, and Steven Cushing for helpful consultations.

We would also like to express appreciation to Andrea Davis for technical editing, Gail Lopes for the preparation of the figures and diagrams, and Mary Yontz for help in the preparation of this report.

M. Hamilton and S. Zeldin

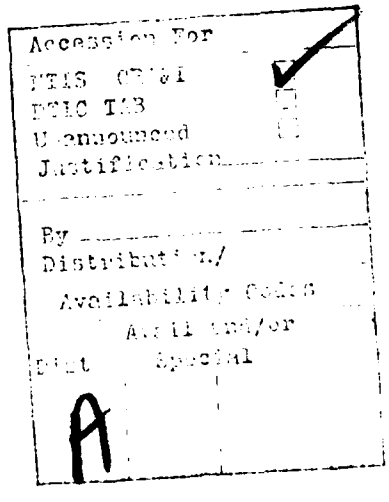


TABLE OF CONTENTS

1.0 MANAGEMENT OVERVIEW

1.1 Background

Inexpensive hardware, expensive software, large complex systems, and a multitude of other influences have converged very recently to cause some fairly large upheavals in the area of developing systems. In fact, events are happening so quickly that it is difficult for management to know where, how and what to respond to. There is, however, evidence of some commonality in the reactions that are taking place and noticeable trends appear to be developing. A sufficient number of large complex systems have been developed or have been attempted to be developed, in which significant problems have arisen to arouse concern.

One very significant reaction to this phenomena is that of DoD who, in turn, reacted with various directives. These directives have had major influence throughout the industry, both in the government and commercial environments. As a result, RFP requirements, contractor requirements, amount and types of customer visibility, contractor qualifications, systems development models, final system products, and methods of producing systems are experiencing major changes.

Many people involved in large systems are beginning to talk to each other, realizing that their problems are not only not unique, but perhaps they can learn from each other in attempting to address the more serious issues that have come to the forefront. For example, software often does not satisfy the original specifications. The basic reason for this is the inadequate techniques that are used for specification--in many cases it would be impossible to develop a correct program from original specifications, for to do so would be like deriving a consistent model from an inconsistent theory.

Our experience has indicated that interface problems (i.e., data and timing conflicts) within a system, between systems, and between various stages of system development account for the majority of the problems involved in the construction of large systems [4]. These interface problems

either took place when attempting communication or resource allocation (a process of preparing for communication).

As a result, the very basis of the methodology of Higher Order Software (HOS), based on our analysis of problems relating to the development of large systems, e.g., Apollo, Shuttle, etc. concerns itself with the definitions of systems so as to eliminate data and timing conflicts.

Not only did we find that interface problems contributed towards making software systems unreliable, but they also increased the frequency of cost overruns and missed deadlines, for such conditions usually resulted when integration of individually completed modules was attempted. No matter how "structured" and correct an individual module may be, unless the system structure is consistent and complete, a project will undoubtably have errors.

Managers are beginning to realize that they, themselves, are in a very enviable position to do something about helping to advance the technology of developing systems, since they, and only they, are the ones who have been forced to lead the way. Upon observing themselves and others, they have come to realize that certain basic concepts such as understanding a problem before solving it are of major importance. Towards this aim managers are recognizing the importance of communication and are now concerning themselves with finding various methods to better define specifications as well as to expedite their definitions and their implementations. As a result, there is now a proliferation of "methodologies" which brings back memories of the proliferation of higher-order languages in the sixties. Within this environment, certain philosophies are beginning to be more commonly accepted. These include the importance of hierarchical decomposition, emphasis on front-end system design, integration of "modules" within a development phase and throughout a development process, and the emphasis on finding or developing an effective requirements or specification language.

There are some concerns, however, in attempting to improve current methods of developing systems. We include here a set of typical questions which reflect these concerns, and answers based upon our own experience. In later sections of this report, we will attempt to address these issues within the context of the PLRS project.

Question: How can we tell if a methodology will work better than no methodology at all?

Answer: Compare the properties of the methodology with those used in an existing development with respect to a well defined set of requirements for consistency and completeness.

Question: How do we choose between one methodology and another methodology?

Answer: Compare the properties of the two methodologies with respect to a well defined set of requirements for consistency and completeness.

Question: What is the difference between using a methodology and using "smart" people?

Answer: The smartest person, by definition, would apply an effective methodology. An effective methodology would far exceed the advantages of a smart person applying his techniques in an ad hoc manner, since all the intricacies of a complex system are by its nature beyond the grasp of one human being. The designs of all smart people must be integrated.

Question: How do we use a methodology without impacting deliverables of an on-going project?

Answer: Choose those aspects of the methodology which find errors or which expedite the design and implementation process.

Question: How do we convince management, designers, and users to use different approaches?

Answer: A different methodology should be demonstrated within the environment of the people who will use it.

Question: What creativity is left for the engineers if a methodology has constraints?

Answer: An effective methodology should support creative designers and not constrain them from producing better designs but rather constrain them from producing errors.

1.2 Existing Methodologies

Although the recently founded philosophical goals of various systems managers are important ones, there exists a proliferation of problems in the attempt to reach these goals, both in developing a new methodology or in choosing an existing methodology. There are, of course, many methodologies whose intent is to solve various aspects of the problem of developing systems. The developers of these methodologies are all proponents of reliable systems with efficient methods for developing these systems. And, most methodologies advocate many philosophies that are similar. For example, it is a commonly accepted idea that it is beneficial to produce a hierarchical breakdown of a given design in order to provide more manageable pieces to work with. And, there are variations between methodologies. Some emphasize a concentration on data flow as opposed to functional flow [10], [1], [3] [12]; others emphasize documentation standards [9] [15]; others emphasize graphical notations [13]; and still others emphasize semantic representation [17].

There are certainly positive aspects in many of these methodologies and, in particular, in what they are trying to obtain. To be effective, however, a methodology should have techniques and rules for the purpose of defining systems which are consistent and complete. But these techniques and rules are useful only if they are within themselves consistent and complete, both with respect to each other and to the systems to which they are being applied.

Too often the same problems exist in the development of methodologies as exist in the problems the methodologies are intended to address. That is, there are often inconsistencies within a methodology. In addition, improvements to a methodology are often ad hoc and modifications to a methodology to fix or enhance that methodology are made to already existing modifications.

Likewise, in the attempt to select an existing methodology, there is always a risk of comparing (1) techniques addressing very different problems, (2) techniques intending to address a problem, but not effectively addressing it at all, (3) techniques with respect to non-existant or ill-

defined requirements, (4) the "syntax" of methodologies instead of the "semantics" of methodologies, (5) techniques, based on unfamiliar paradigms with preconceived notions, (6) techniques addressing the wrong problems or those which are "in the noise," and (7) techniques with respect to completion or amount of use rather than with respect to the problems they are solving.

1.3 Requirements for an Effective Methodology

Most importantly, the time has come when one is forced by large systems to look closely at properties of systems. They are more basic than one cares to admit. If this fact is ignored, there is a risk of responding to only symptoms and investing a great deal of effort based on preconceived notions and misquided misconceptions.

In choosing requirements for a methodology, issues such as how people think, learn, communicate, and resource allocate need to be addressed. These issues are not unlike those of "older," more established fields, like philosophy or mathematics, and more recently, linguistics. But when working with large systems, there is the advantage of more visibility into some basic issues than was ever provided before. What is suggested is not only a whole new set of paradigms for developing systems, but more importantly, a whole new attitude on accepting that fact. It is within the framework of such a set of paradigms that proper research requirements for methodologies, including methods for specifying specification techniques before specifying systems, can be determined.

The first step, then, is to define more explicitly what it is that needs to be solved and then to define more explicitly how to respond to this need. To be effective, a methodology should have the mechanisms to consistently and completely:

> Define an object and its relationships formally. That is, every system in the environment of an object system (people, hardware, tools, software) understand a definition of an object and its relationships the same way.

Provide for modularity. That is, any change should be able to be made locally (with respect to levels and layers of development [5]), and if a change is made, the result of that change should be able to be traced throughout both the system within which that change resides and throughout other systems within that system's environment.

Provide a set of primitive standard mechanisms which are used both for defining and verifying a system in the form of a hierarchy.

Provide for an evolving set of more powerful (with respect to simplicity and abstraction) mechanisms based on the standard set of primitive mechanisms.

Allow system engineers to communicate in a language (with common semantic primitives and a dialect of their choice) which is extensible, flexible, and familiar and which serves as a "library" of common data and structure mechanisms.

Provide for a development model which includes a set of definitions, tools, and techniques which support a given system development process.

1.4 How HOS Responds to the Requirements for an Effective Methodology

We will discuss HOS in terms of the requirements we have set forth for methodologies in general, before discussing the application of HOS within the environment of the PLRS project.

FORMAL

HOS systems* are formal in that the relationships of all objects are explicitly defined in terms of completeness of control. That is, all HOS systems always have the same properties with respect to control of interfaces as a result of standard and well understood ways of defining interfaces. Thus, everyone defining a module, using HOS, must follow the same rules as everyone else in constructing the structure of that module. The control of every object in a system is determined by adherence to six axioms [c.f. Appendix I].

*by a system we mean an assemblage of objects united by some form of regular interaction or interdependence, where an object is an existence of something.

These axioms are, in essence, a consistent set of rules that determine a means for defining systems that are consistent, complete and not redundant. These rules determine a means for defining invocation of functions, input, output, input access rights, output access rights, error detection and recovery, and ordering of functions. When these rules are applied, there is no room for ambiguity with respect to control. That is, everything must be controlled and every object has a unique controller. All objects in an HOS system can be described in terms of control structures, derived from the six axioms, that relate members of algebraicially defined data types or functionally defined data whose components are algebraically defined.

MODULAR

Systems defined in HOS satisfy the requirements we have set forth for modularity. Control, or the chain of command, can be traced directly on an HOS control map. Function flow (including both input and output) can be traced directly on an HOS control map. In addition, the nature of HOS systems is such that the mechanisms of defining systems as well as the systems themselves, behave as if they are "instructions," e.g., a given control structure has no knowledge about a higher-level control strucutre. With these properties, changes can be traced directly and changes can be made locally. Systems defined in HOS display certain other distinctive properties. For example, HOS systems have been shown to be secure systems [2] and the single reference, single assignment properties of HOS systems provide an interesting set of resource allocation alternatives.

PRIMITIVE STANDARD MECHANISMS

Three primitive control structures, derived from the axioms [4] provide rules for the definition of dependent functions (e.g., sequential processing), independent functions (e.g., parallel processing), and selection of functions (e.g., reconfiguration). From a combination of primitives, more abstract control structures can be found (e.g., recursive

functions). A complete design is one which has been hierarchically decomposed until all terminal nodes of a control structure represent primitive operations on data types.

Thus AXES, the specification language based on HOS, is able to have a common set of specification primitives (i.e., a common specification "machine"). As a result it is possible to have common tools, such as an analyzer to check for correct interfaces (i.e., completeness, consistency, and elimination of redundancy) and a resource allocation tool to prepare a specification for a particular machine environment (c.f. Appendix II).

EVOLVING MECHANISMS

Although a system can be defined directly with AXES, a more powerful use of AXES can be made by defining systems which are themselves mechanisms for defining systems. Thus, we can define a set of specification "macros" which collectively could form a "language" (or management standards) for defining a particular system or family of systems. Each new system user is able to use a subset of already defined statements in an AXES-based library or add new statements since the AXES language system is extensible both with respect to structure and data definitions.

FAMILIAR DIALECTS

AXES provides a user with the capability of defining any syntax desired for a control structure or data type. Thus, for example, a communications project is able to have its own set of specification statements to use as a means of standardizing and an avionics project is able to have its own set of statements. But, both are able to communicate in common at the level of the primitive specification machine to which these statements can be reduced to their primitive form.

DEVELOPMENT MODEL

AXES provides the mechanisms to define a development model as a system and to define the management of a system development model, which uses that development model, as a system. Within the context of a complete development provess, HOS provides a means to define management standards, definitions, milestones, disciplines, phases, and tools and techniques and the relationships between all the various components within a development process.

1.5 Previous Experiences with the Application of HOS

Once the foundations of HOS were formulated, it was then necessary to apply the methodology to some actual applications in order to demonstrate its effectiveness.

Initially, we chose the Apollo Guidance Computer (AGC) operating system, an application familiar to us [8]. Unfortunately, we had a great deal of difficulty reconstructing the pieces. This was due mostly to the fact that the AGC operating system was poorly documented. Our only solution for completely understanding the system (which included our own design and our own coding) was to go back and pour over the original code, which was very clever and difficult to understand. When we began this effort, we thought there was little in the AGC operating system we could improve upon. This attitude was partly as a result of the fact that no errors were found for several years within the operating system (OS) itself. However, when we attempted to specify the operating system with HOS, we discovered that many of the development errors which occurred in the application programs, using the OS, would not have occurred if the AGC OS had certain other inherent properties. For, although the AGC OS had properties of hidden data, it did not have properties of hidden timing. From this effort, we therefore determined that HOS was very helpful in demonstrating more reliable design goals with respect to interfaces between application programs and the systems software which executes these programs.

We then selected an application to demonstrate another aspect of our techniques. Here we extracted a problem definition from an existing description of a typical orbit/altitude spacecraft problem to demonstrate the abliity of our technique as a guide to design and verirication [6]. Although this problem was relatively small, we were able to use HOS in determining what questions to ask in understanding the problem. In addition, we provided alternative designs at the level of the user interface, so that the human user functions were less error prone. Many of the interfaces existed only in the minds (as is characteristic of most projects today) of a small collection of experts who had been involved in the original project. Our emphasis was to integrate and fill in, where necessary, interfaces that were missing in the specification document.

We have just completed the definition of a multilanguage structured flowcharter in AXES which we are implementing in PASCAL [7]. The programmers who are implementing the flowcharter are determining the design of the code by using the control maps as a guide to design. From the control maps, the programmers are able to directly determine the "whats" with respect to implementation and the alternatives with respect to the "hows" of implementation.

The above three tasks contributed in the determination of the effectiveness of HOS with respect to real world applications. We had not, however, attempted to apply HOS to an ongoing project, or to a project whose application was quite unfamiliar to us. Such an opportunity was provided to us by PLRS.

1.6 An Overview of the PLRS Project

Our charter, with respect to PLRS, was to select the most complex module, specify that module in terms of HOS, and demonstrate the advantages of applying an effective methodology (in particular, HOS). We did just that. But, there were several additional interesting results and observations that resulted from this effort. A detailed description of this effort is contained in the remaining sections of this report. The following

summarizes the PLRS effort and the key results of this effort.

- The most complex module of the network manager has been specified with an HOS control map.
- Identification of commonality between modules was shown.
- A description is provided to show how we defined the network manager control map.
- Differences between the control map and the information provided in the PPS have been determined.
- Advantages of control map techniques with respect to management, design, implementation, verification, and documentation have been determined for PLRS.
- Sixteen categories of questionable areas such as unanswered questions, inconsistencies, incompleteness, and redundancies have been determined.
- Some suggested methods of specifying the control map with AXES statements are shown. A comparison of the PPS with an alternative method using AXES is provided.
- Specific recommendations with respect to the network manager have been determined.
- Standards (common structures and data types) have been defined for the network manager with AXES. These standards can be used not only for other PLRS modules but for a family of communication systems as well.
- A section of the PPS was rewritten to incorporate HOS techniques in order that a comparison could be made on a one-for-one basis between the existing PPS and a PPS using AXES, the specification language of HOS.
- General recommendations with respect to the PLRS project have been determined.
- General recommendations with respect to further efforts have been determined.

1.7 PLRS: Lessons Learned

PLRS is the first effort in which we attempted to use HOS on an ongoing project. Not only was our aim to demonstrate the effectiveness of HOS, but also to perform this task without impacting schedules or deliverables of the PLRS project. In this process, however, we determined that the use of an effective methodology can not only benefit a new project, but it can also benefit an ongoing project which already employs a different methodology. Such benefits, some of which are described below, fall into two main categories: those which make the system more reliable and those which help to accelerate the development process.

Acceleration of the Learning Process

The people who performed the HOS/PLRS task were unfamiliar with the PLRS project. This has its advantages and its disadvantages. The disadvantages may appear obvious, for it is always helpful to understand as much as possible about an application before working on it.

But, because we were unfamiliar with the PLRS effort, we were able to take advantage of such a fact in order to test HOS as a learning technique. Our method of understanding the network manager was to first attempt to construct a control map and by doing so, we were able to determine existing functions and their relationships. This process not only provided us with an accelerated means of asking the questions that should be asked to construct the definition of the network manager, but it also became clear that this was a technique for prompting questions that otherwise may never have been asked. For, during this process, we found that there were areas in the PLRS documentation which were not clear enough, missing, inconsistent, redundant, or not integrated with other areas.

The fact that we were able to use the HOS control map technique as an accelerated learning process for ourselves suggested to us that this same technique could be used as a learning tool, for example, for those

new people coming aboard a project, a manager learning about the work of the people in his project, designers learning about each others modules in the same project, implementers learning the specification they are implementing, and users, such as maintenance people, learning the system they are using or changing.

Acceleration of the Specification Process

Although the specification of the network manager (the particular PLRS module chosen for demonstration) was, for all practical purposes, thought to be complete, it was necessary for us to design more explicitly function definitions, including data definitions, as well as the integration of these functions. In the process of constructing the various components of the network manager, the HOS control map technique was quite effective in expediting design processes. By using the control map technique we were able to determine:

- Types of design tradeoffs
- Correctness of design decisions with respect to consistency, completeness, and lack of redundancy (i.e., verification before the fact).
- Common use of specification modules (data types and structures)
- More powerful and simpler ways of conveying specifications
- When each specification module is completed
- How to safely integrate all the modules in the system
- Common rules (or management standards) of communication between modules in the system
- Methods of defining the system so that changes could be made safely and the effects of those changes traceable within the design and during the design process.

It was clear in the PLRS effort that the HOS methodology not only supported a designer in providing designs more quickly, but it also helped to point out things he might have forgotton about completely.

Verification and Validation Aid

In the process of analyzing the network manager and redefining its most complex modules with HOS, we were able to show the effectiveness of HOS as a verification and validation aid. Several errors were discovered by the two step process of (1) formally defining the data types that were used and (2) formally defining the structure (or organization) of the network manager. These errors were found by checking existing specifications from the standpoint of interface reliability using control structure and data type mechanisms. All in all, sixteen categories of questionable areas were found. If problematic areas are detected early as illustrated by the application of the control technique to PLRS, later development phases can benefit, since problems are not only able to be detected earlier or prevented before the fact, but these problems will not surface later or propagate into worse problems.

Establishment of Design Goals

In the process of understanding the network manager, it would have been helpful if the PPS had concerned itself more with the definition of how the specified functions related to each other (particularly at the top level). The control map technique forced us to consider integration of the functions of the network manager from the very beginning. Such a design philosophy, if applied, not only aids in understanding a design but eliminates integration problems that would subsequently show up in later development stages, such as the PDS. Thus if the PPS was integrated the PDS could be an evolving document in stead of a "redo" of a more detailed PPS.

Enhancement of Existing Techniques

We were able to indicate certain problem areas or demonstrate ways of making certain improvements to the PPS without impacting schedules or milestones. Advantages can be taken of our findings such as I/O compatability and understanding the data involved, within the environment of previously existing techniques other than our own.

Management Visibility

During this project, we were able to determine a "feel" for the state or health of the specifications of PLRS, in general, by viewing a section of PLRS. That is, we were able to get a better idea of the types of interface problems that needed to be resolved before the specification could be successfully implemented. We were also able to determine what steps would be necessary before the specification could be called a complete specification. And we were able to determine certain recommendations which we thought would be quite helpful in providing a more reliable specification more efficiently in the future.

1.8 Recommendations for PLRS and Future Efforts

Put simply, the most urgent need on any large system development process is that of standardization. Ultimately, aside from being effective, standards should be consistent with each other, not redundant, and complete.

Some standardization, if it is effective, is certainly better than none at all. But, if a project is already in development, it is not usually possible to apply an ideal and complete set of standards. But it is possible to incrementally begin to use those standards which would enhance the development process either by finding errors or by accelerating remaining phases of development. We did this on Apollo. For example, we discovered that many interface errors took place in the implementation phase when programmers would use instructions like "GOTO +3." Errors would creep in when someone would come along, often the same programmer, and inadvertantly insert a card between the GOTO instruction and the location it should have gone to. Once we discovered the amount of errors which resulted from this use of our language, we enforced by standardization the use of instuctions such as "GOTO A" rather than "GOTO +3." As a result, errors which fit into the above category never happened again. This type of incremental enhancement to our own methods was very effective on an ongoing project. The same sort of introduction of standards could take place on PLRS and other projects, for PLRS is not unique in requiring certain enhancements. If anything, the PLRS documentation is quite representative of documentation we have reviewed from several projects; for we all suffer

from the syndrome of hurrying to get the design process done because of deliverables which appear impossible to meet, especially if we pause to come up with standards. But hindsight and recent experiences of our own and others have demonstrated that in the end it pays to organize first and build later, particularly when we are involved in the development of large and complex systems.

There are several standards that we recommend be used in the PLRS or a PLRS-like environment:

- Definition of design goals - e.g., definition of interfaces should be made in the PPS phase, i.e., integrate from the beginning.

- Rules for design and verification - specifications should be defined hierarchically and rules (e.g., those that accompany the control map) should be followed with respect to how one level in a heirarchy relates to the function directly above it. These rules should include ways of defining the invocation of a set of functions, input and output flow, input access rights, output access rights, error detection and recovery, and ordering.

- Interface Specification Document - for every system a standard dictionary (or library) should exist which provides common meanings, ways of saying things, ways of doing things, mechanisms for defining a system, system modules, and support tools and techniques. For PLRS, we would have found it extremely useful, and believe the PLRS people could benefit even more, if an evolving dictionary were introduced which included a set of
  - definitions of terms
  - formally defined data types
  - formally defined control structures
  - system functions

- User Manual - a user manual should be provided which contains checklists and explains (1) how users interpret the standards in the interface specification document, (2) how designers design modules to add to the "library" of the interface specification document; and (3) how managers define new standards for system development which in turn can be converted to modules, by the designers, to incorporate into the interface specification document.

- User Guide to Implementation - If specifications contain certain consistent properties, one can take advantage of these properties by understanding their consequences with respect to implementation. Given, that there are standards for specifying, it would expedite the implementation process if standards were defined to go from a specification to an implementation. The user guide should include standards for (1) going

from the specification (e.g., a control map) to a computer allocation; (2) reallocationg functions to a computer, and (3) providing for reconfiguration of functions in real time.

- Definition of Development Model - the definition of a development model is most helpful to the manager who is responsible for integrating all the phases of development. In addition to the above recommendations, the development model should define phases of development and how to integrate them, disciplines (such as management, design, verification, implementation, and documentation) and an integrated application of tools and techniques that are to be used, how they are used and when they are to be used throughout the development process.

1.9 Implications and Payoffs for the Future

In order to change to new techniques, there is always the initial investment that is necessary for defining and developing a model (or subsets thereof) for systems in general. We believe that a great deal of work necessary for this step has already been accomplished within our own methodology.

A next step is to define a set of additional structures and data types that are necessary for defining a particular family of systems (e.g., PLRS is a member of a particular family of communications systems). Once the initial investment has been made to establish, what in essence, is a way of organizing the development of a system with standards and mechanisms to accomplish that organization, the payoffs should be quite apparent. Design time during the requirements/specifications phase should be no greater and, in fact, we suspect much less than with current techniques. Implementation designs should take considerably less time than with current practices since it is possible to perform such a process on an almost one-for-one basis. We suspect that the largest savings will be realized within the verification processes since most of the recommended techniques provide standards which should eliminate errors before the fact and it is just these very types of errors that we spend so much time looking for today.

2.0 THE PLRS PROBLEM

The Real Time PLRS system [11] [14] is a combined communications and ranging (position location reporting) system, in which many radio User Units (UUs) are deployed in a field of operations, carried by hand, or mounted in planes, helicopters, tanks, etc. The User Units communicate with one another and with a stationary Master Unit (MU). Transmissions are relayed to and from the Master Unit along a series of communications links called PORT Links (Figure 2-1). The User Units passively receive transmission from certain other User Units for the purposes of determining location, and the Master Unit computes the position of each User Unit from these measurements and displays it to the human operator. Such passive links are called CROSS Links. The network, consisting of these PORT Links (forming PORT Paths to the Master Unit) and Cross Links, is continually being reconfigured as User Units enter the network or drop out, as certain links become unreliable, as geographical configurations of units change, etc. As one part of the Real Time PLRS system, there is a module called the Network Manager (NM) whose function is to supervise the continual reconfiguration of this communications network. This report discusses some aspects of the specification of this module as an illustrative example of Higher Order Software (HOS) methodology.

2.1 The Need for Formal Specifications

In the sections that follow, we attempt to give a specification using HOS for part of the Network Manager module of the Real Time PLRS system, consisting of control maps, algebraic data-type specifications, and sample AXES language statements. We would like the designer to communicate his design in a precise, uniform way, as well as make changes without having committed either hardware or even detailed software; for the user to review and verify the specification in detail and propose revisions; for the programmer to receive his instructions in an unambiguous way, so that time is not wasted in trying to resolve conflicts in the specification; and for the manager to allow the flexibility of reusing the same specification in different situations which may require different hardware commitments or where different implementation systems may be available.

THE UU COMMUNITY AS A NETWORK

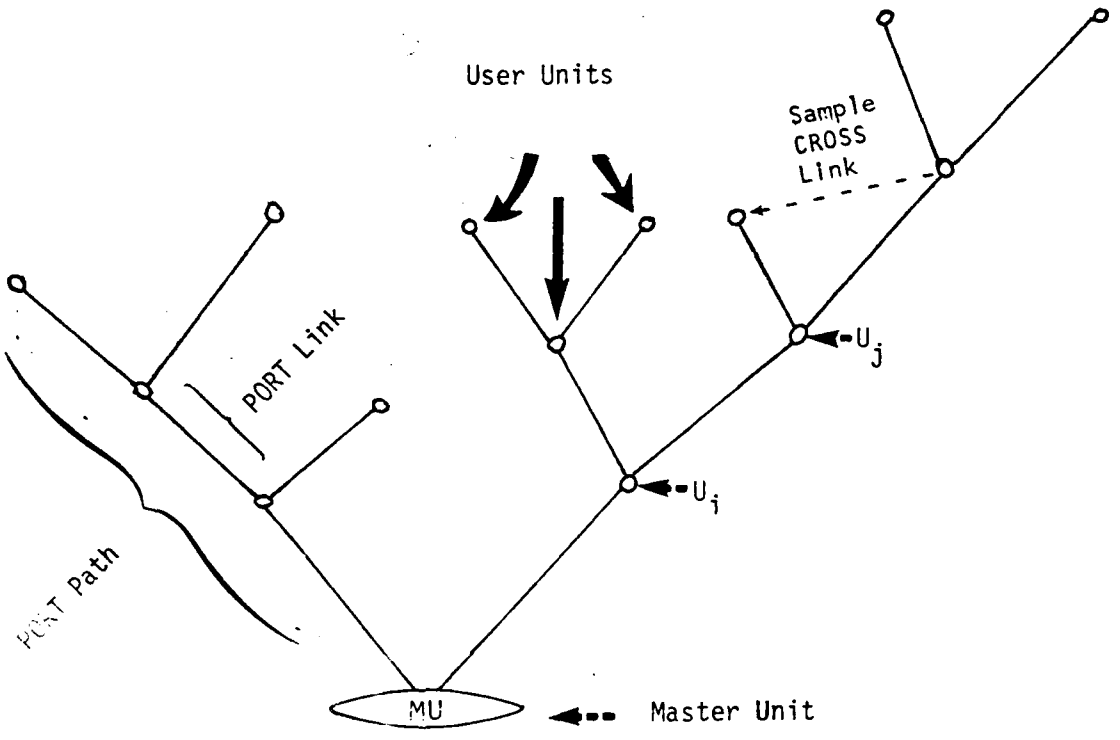


Figure 2-1

This exercise, then, is to take a section of an actual system, the Hughes Real Time PLRS, and, using information gained from some of Hughes specifications as well as discussing the problems with people at Hughes and Ft. Monmouth, plus making some hypothetical assumptions, to demonstrate how HOS would specify the system, by specifying a small part of it.

A way to demonstrate what is missing from, for example, the Program Performance Specifications (PPS) [11], is by presenting a trivial example in the same style. We might try to specify a hypothetical module, the HOUSEKEEPER, which describes some of the functions of a person living in a house (Fig. 2.1-1). This is presented in the style of the PPS, which also accompanies the verbal descriptions by diagrams like those in Figure 2.1-2. If we were to take this analogy seriously (for example, as a class exercise in flowcharting for a beginning computer programming class), we already note some glaring inconsistencies. That is, there are

(1) ill-defined functions and/or operations,
(2) syntactic ambiguities,
(3) unspecified assumptions,
(4) lack of hierarchical structure (how it fits together),
(5) inconsistencies, especially in I/O interfaces,
(6) areas where control could be lost by the software operator,
(7) redundancies.

Although we may assume some of I/O interface inconsistencies are typographical errors, these problems are basically problems with this style of exposition, which may reflect possible problems in the software it represents. In Figure 2.1-1, there are examples of (1)-(7) as follows:

(1) Sect. 3.4.2.1.2.2. MAKING SUBFUNCTION. "Making" dinner and "making" beds are not only not the same operation, but have nothing to do with one another.

(2) Sect. 3.4.2.1.2. STRUCTURE OF THE HOUSEKEEPER. Washes what? Dishes? Floors? Furniture? Dries dishes? Dries furniture?!!

A Whimsical Analogy

3.4.2.1. Housekeeper Function: The function of the housekeeper is to keep the house.

3.4.2.1.1. Inputs to the Housekeeper Function: The inputs to the Housekeeper Function are those listed in Appendix J. [We turn to Appendix J and find listed "Brillo Pads, mop, dishcloth, laundry soap, Ajax, water, dustmop, dish soap,..." in that order.]

3.4.2.1.2. Structure of the Housekeeper: The Housekeeper washes, dries, dusts, shops, and cleans dishes, floors, and furniture and makes beds and meals.

3.4.2.1.2.1. Cleaning Subfunction: The Housekeeper cleans several objects and rooms.

3.4.2.1.2.1.1. Washing Dishes: The Housekeeper uses dish soap and water to wash dishes.

3.4.2.1.2.1.2. Drying Dishes: The Housekeeper uses a dishtowel to dry the dishes. [NOTE dishtowel not listed in Appendix J!]

3.4.2.1.2.1.3. Scrub Bathtub: Housekeeper uses Ajax and toilet brush (?!) to scrub bathtub.

... Etc.

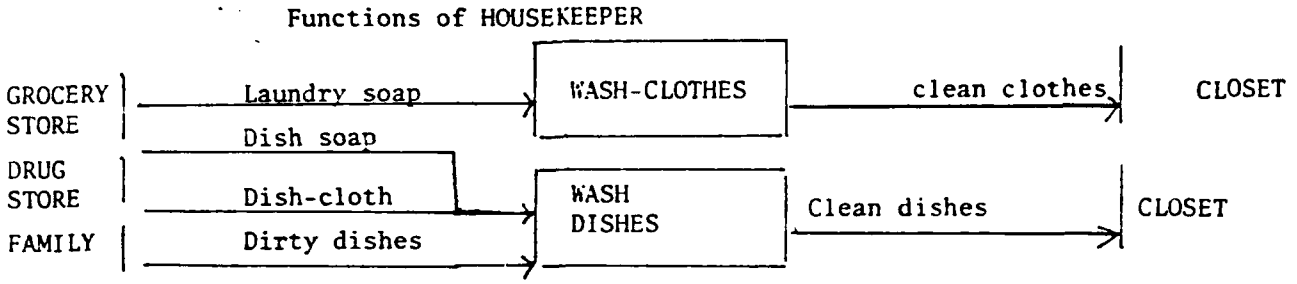
3.4.2.1.2.2. Making Subfunction: The making subfunction makes dinner which is output to family, as well as making beds which are located in several rooms. The making subfunction depends on the Shopping module.

... Etc.

3.4.2.1.2.8: Shopping Subfunction: The Housekeeper buys various supplies to maintain inventories at a safe level (a system parameter). No more money is spent than Budget (a system parameter).

... Etc. ...

Figure 2.1-1



There are, in addition, operation tables:

	GATHER	WASH	DRY	STORE
Wash-dishes		X		X
Wash-clothes	X	X	X	X

Figure 2.1-2

(3) Sect. 3.4.2.1.2.1. CLEANING SUBFUNCTION. Which objects? Which rooms?

(4) What is the relation, timing and otherwise, between kitchen (cooking) functions, cleaning functions, etc.? For example, washing and drying dishes presumably follows, not preceeds, making dinner.

(5) Sect. 3.4.2.1.2.1.1. No dishcloth is listed in Figure 2.1-1 as input to WASHING DISHES, but <u>is</u> listed in Figure 2.1-2 as input.

(6) Suppose the housekeeper goes out shopping to obtain food (i.e., "inputs")and meets a friend instead, goes to movies, etc. and doesn't come back to make dinner.

(7) Sect. 3.4.2.1.2.1. CLEANING SUBFUNCTION is not only not a unitary function, but adds no information not already contained elsewhere.

Similarly, we find examples of problems (1)-(7) in the PPS:

(1) What is PORT Link Assignment? Since this is the central concept in the Network Manager, it seems odd that it does not appear in the PPS glossary. The closest thing is ASSIGNED PORT LINK: "a communication link defined by a set of active frames specified by a PORT Link Assignment," which is circular. Also, as will be discussed later in this report, the PORT Link Assignment function itself is ill-defined, in that it is two-valued, without specifying which of the two PORT Link Assignments is being referred to at any given point. (This is discussed in Section 3.4.)

(2) The draft PPS contained the sentence shown in Figure 2.1-3a, which we interpreted as containing the two underlined noun phrases, and spent a good deal of our time in trying to determine what a "processing link" was. The correct interpretation

of the sentence is shown in Figure 2.1-3b. This is an unavoidable property of natural language, which is eliminated by using formal, mathematical specifications.

3.4.2.2 Network Management Processing: The Network Management automatically determines and maintains PORT and Cross Link Assignments for each UU as well as processing link and state change requests...

(a)

3.4.2.2 Network Management Processing: The Network Management automatically determines and maintains PORT and Cross Link Assignments for each UU as well as processing link and state change requests...

(b)

Figure 2.1-3

(3) The section from PPS shown in Figure 2.1-4 contains a surprising bit of information, which we were unable to locate elsewhere. Evidently, for each User Unit, the number of other User Units which have tried to gain entry to the system through that User Unit and failed is a parameter being stored and continually updated.

3.4.2.2.1.5 Monitor Entry Requests: A UU that has been a two-way communicant with more than the unsatisfied entry count (system parameters) entry requests of UUs, for which a PLA was not found, shall be designated for PORT Link state change.

Figure 2.1-4

(4) In Figure 2.1-5, we see two different accounts of where ZERO ALERTS are generated. In the I/O diagrams, it is being generated by PORT Link Assignment Control (PPS, p. 3-35/6), but

in the text, it is described as generated by Network Control (PPS, p. 3-38).

3.4.2.2.1 Network Control....

3.4.2.2.1.1 UU Entry Requests... When a PLA is not found for an entering UU, a zero alert shall be generated for that UU.

BUT

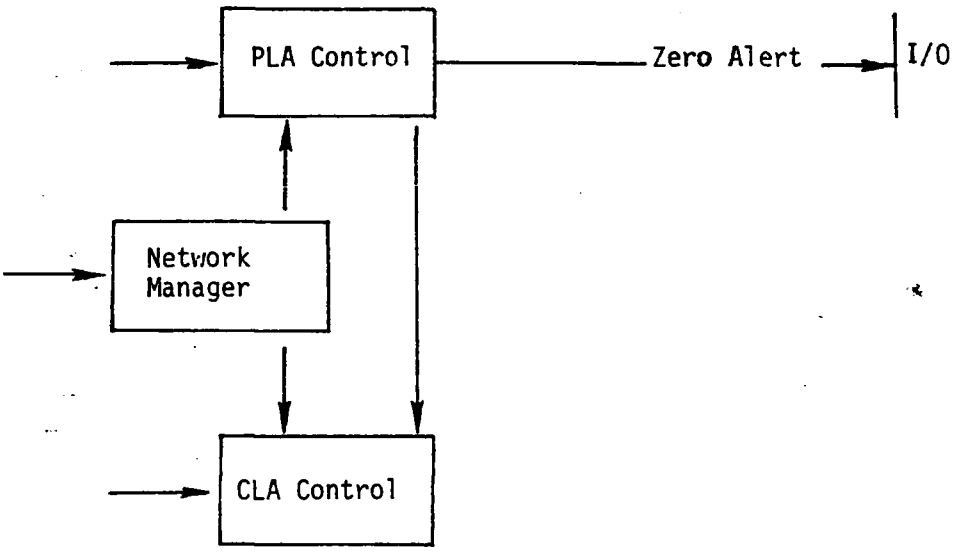


Figure 2.1-5

(5) In the PPS diagram 3.3.5-1 (p. 3-13/14), ALERTS is an input to the Network Manager; Figure 3.4.2-1 indicates it to be an output. The former indicates no output from the Network Manager to the POSITION Tracking Module; the latter indicates "Unlocate track is output by the Network Manager to Position Tracking." There are many discrepencies, discussed at greater length in Section 5.

(6) The example cited in Figure 2.1-6 is similar to the "Housekeeper" shopping example: suppose a HANDOVER-IN SUCCESSFUL notice, for whatever reason, never arrives? What about error recovery here?

> 3.4.2.2.1.1.3 HANDOVER-IN UUs. Handover-in UUs shall be initially designated as being in the zero rate PL state and shall be processed by PLA control. Network control shall generate a tree allocation command containing the MU tree allocation to all handover-in UUs. Upon acknowledgement of this command, network control shall generate a handover-in successful notice.
>
> Figure 2.1-6

(7) As we will see later in the discussion of control maps, statements such as "Each PLA shall be supported by only one UU" are either meaningless, false, or redundant. See Section of this report for a complete discussion of this issue.

How are such problems discovered by attempting to do an HOS specification, and how does an HOS specification eliminate them? This report tries to illustrate this by way of example. Part of the detection of these problems was done in the early stages of attempting the specification, by drawing a preliminary control map along the lines of the Hughes PPS. This was discussed in [16], which is excerpted in Section 2.2 below. Other problems were subsequently detected in trying to formulate the data types and their behavior; some of this was also discussed in [16] . A complete discussion appears in Section 2.2 below. In Section 3 we will discuss a control map for one subsection of the Network Manager.

2.2 Preliminary Control Map

Having reviewed the STD [14] and the final draft of the PPS [11], we began to define a control map. At this stage, rather than doing a control map "from scratch," since the algorithms in many cases were

preliminary control map was done directly from the specification in the PPS, without regard as to whether it followed HOS axioms. This gave the structure of the Network Manager as it is presented in the documentation. By then trying (1) to correct the structure of the control map and (2) make it consistent with the various descriptions of the Network Manager at various levels of representation, we could identify potential problem areas, even though the control map was, at this stage "content free." That is, subroutines were simply treated as lettered subfunctions and the partitions given in the text accepted without regard as to what the subroutines (subfunctions) actually did. This was done later. Even at this stage, however, some problems are identified. For example, the PPS contains three different levels of representation of the input/output structure (see Figure 2.2-1 for exact labeling).

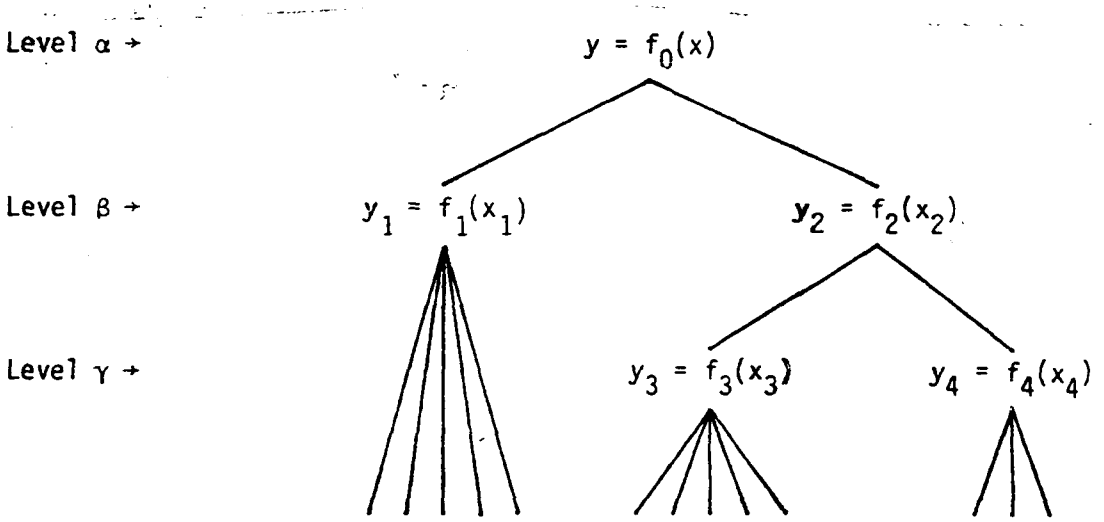


Figure 2.2-1

If we take f_0 to be the Network Manager function, the Figure 3.3.5-1 [11] on pages 3-13/14 corresponds roughly to Level α; Figure 3.4.2-1 on page 3-35/56 [11] corresponds to Level β; and the verbal descriptions of the subfunctions on the pages following page 3-37 [11] correspond roughly to Level γ.

The first thing we notice is that, strictly speaking, this control map should be a Class Partition. That is, if $x_1 = (a_0, a_1, a_2, \ldots a_k)$, then $x_2 = (a_{k+1}, a_{k+2}), \ldots a_n)$, etc. as we go to deeper levels of representation. This is not the case in the Hughes PPS, however.

Sometimes the differences are "trivial" in that the same name is not used for the same data item at all levels. However, while this may seem trivial (like a "typographical error") at first glance, if the documentation is to be relied on, it is at best confusing for manual verification, and it is death for machine verification.

For example, in Figure 3.3.5-1 [11], the I/O inputs UU (User Unit) CONTROL to the NM (Network Manager); in Figure 3.4.2-1 [11], it inputs UU MODE CONTROL. At the lowest level, that of the verbal description, the UU Mode Control Processing subfunction accepts as inputs PASSIVE MODE REQUEST, REENTER UU REQUEST, RESTART UU REQUEST, CLEAR UU REQUEST, REINITIALIZE POSITION TRACK REQUEST. It is not clear which of these are part of the data item "UU MODE CONTROL" from the I/O and which are not. While these are discrepancies in the documentation (which would hopefully be more precisely specified enroute to coding), the HOS control map allows one to see immediately which inputs at different levels of specification must be identical. Thus, one can prevent inconsistencies before they happen.

Thus we see two kinds of problems right away. As we go from, for example, Level_i to Level_{i+1}, either we must have the same set of input variables (Figure 2.2-2)

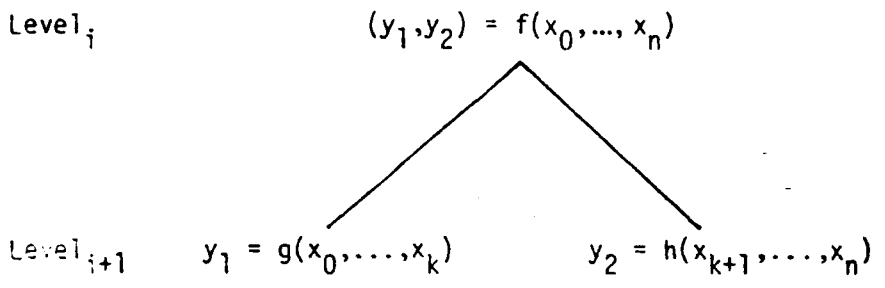


Figure 2.2-2

or, if one wished to abbreviate for compactness of presentation, one would have to be very careful to specify which input variables at $Level_{i+1}$ were represented by which variable at $Level_i$ (Figure 2.2-3):

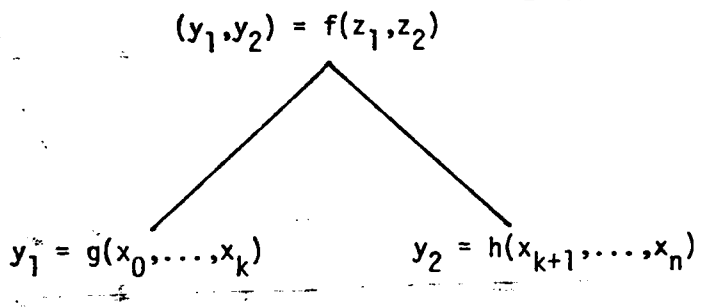


Figure 2.2-3

where $z_1 = x_0,\ldots y_k$ and $y_{K+1},\ldots x_n$. This may seem unnecessary for such a simple partition as the one above, or even a small part of the NM, especially at the top levels (0,1,2, etc.). However, in the more detailed specifications (lower levels of the control map tree), the chances for error multiply rapidly without keeping this in mind. The control map and axioms force one to be consistent from top to bottom. This is even more apparent if one thinks of a PLRS Master Unit as being $Level_0$, and the NM as being one section of a lower level.

Another area in which the control map can be helpful is in identifying where the same data items (or subparts of the same data items) are being input to two different modules. For example, in the accompanying preliminary control map done from the PPS (see Figure 2.2-4), $x'_{14} = x'_{21}$ = COMMUNICANTS, $x'_9 = x'_{15} = x'_{22} = x'_{24}$ = COMMAND-ACKNOWLEDGE. This is partly simply a matter of clarity and perspicuity. However, one then wants to ask whether they are to be treated as unitary items, or whether they are being input to several functions implies a partition, as in the Class Partition exemplified above. In terms of the control map, for example, the input from the Master Traffic Control (MTC) at level α, x_{13} = COMMAND RELIABILITY (from [1] Figure 3.3.5-1) is presumably partitioned into

PRELIMINARY CONTROL MAP FOR THE NETWORK MANAGER

(done from the Hughes PPS; Includes specification of one sample subfunction at Level γ)

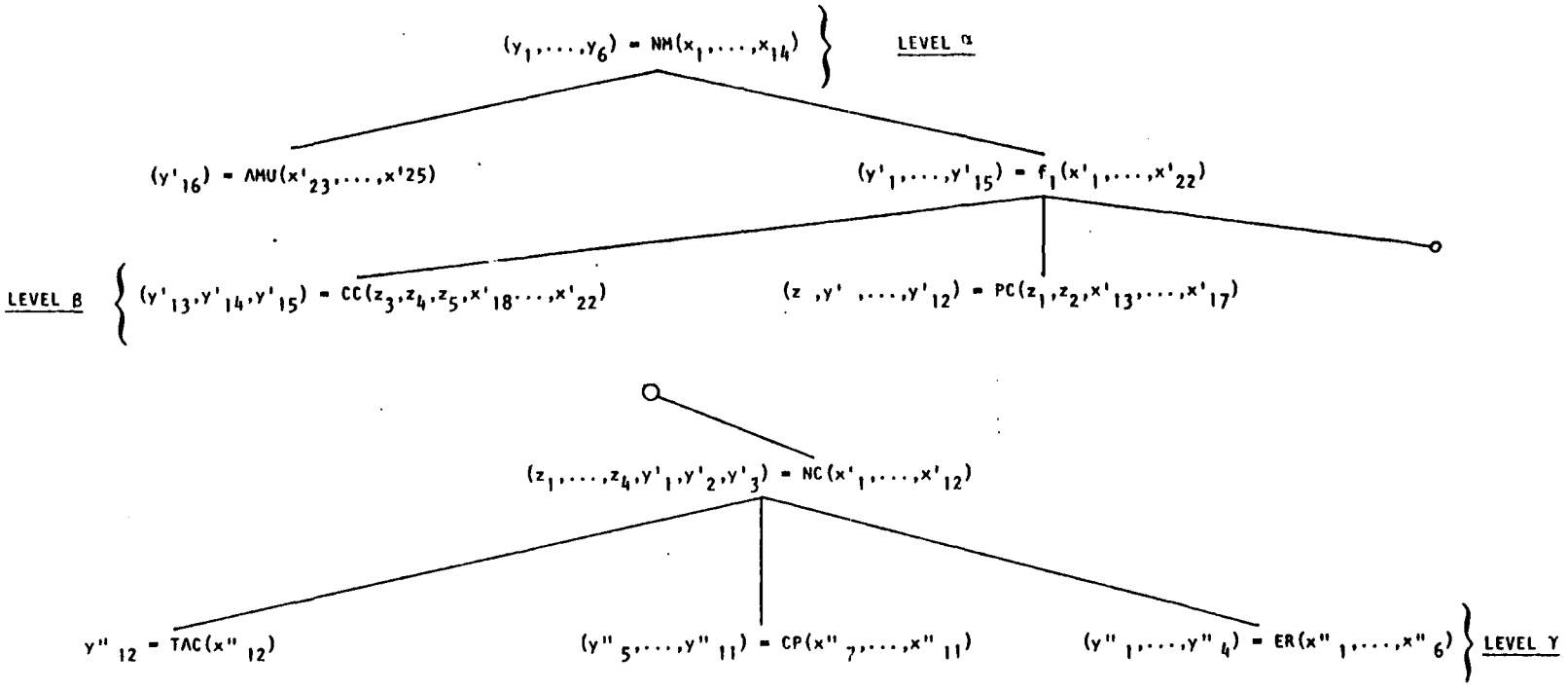


Figure 2.2-4

PRELIMINARY CONTROL MAP for the Network Manager

Key to function names:

NM = Network Manager

AMU = Adjacent MU Communications Assignment

f_1 = (arbitrary label for node)

NC = Network Control

PC = PLA Control

CC = CLA Control

ER = UU Entry Requests

CP = UU mode Control Processing

TAC = Tree Allocation Change request

[NB: One the following pages are the keys to the variable names x_1, x_2, etc. Since this Preliminary Control Map was done literally from the Hughes PPS, the x_1, x_2, ... are not to be identified with the x'_1, x'_2, ... or the x''_1, x_2'', ..., although in a proper Control Map they would be the same. Part of our task is to identify which x_i = which x'_j; this is precisely what is unclear in the PPS.]

Figure 2.2-4 (con't)

NETWORK MANAGER: Preliminary Control Map. Input/Output variables, Level α.

INPUT,
FROM
↓

OI: x_1 = Alerts {should be output}
x_2 = Track Control
x_3 = UU (Mode?) Control
x_4 = Link Control

PT: x_5 = Link value

SU/SO: x_6 = Polling list
x_7 = PLA

RPT: x_8 = Unlocate track
x_9 = Set passive
x_{10} = Handover
x_{11} = Cycle rate

MTC: x_{12} = UU Entry
x_{13} = Command reliability
x_{14} = Communicants

OUTPUT,
TO
↓

MTC: y_1 = PLA
y_2 = CLA
y_3 = UU mode requests

OI: {none indicated}

RPT: {none indicated}

PT: {none indicated}

PM: y_4 = Network data

SU/SO: y_5 = Polling list {not indicated in β level}
y_6 = PLA

Figure 2.2-4 (con't)

NETWORK MANAGER: Preliminary Control Map. Input/Output variables, Level 8

INPUTS

from: ↓	to NC	to PC	to CC	to AMU
OI:	x'_1 = Track control x'_2 = UU (Mode) control	OI: x'_{13} = Link control	PT: x'_{18} = Link value x'_{19} = Located UUs	
SU/SO:	x'_3 = Start-up polling list			
RPT:	x'_4 = UU Cycle rate (change request) x'_5 = UDIO Re-enter req. x'_6 = UDIO Set passive req. x'_7 = UU being handed over x'_8 = Handover-in req.			RPT: x'_{25} = Secondary MU assignment verification
MTC:	x'_9 = Command acknowlegement x'_{10} = (UU) Entry req. x'_{11} = UU type x'_{12} = Re-enter UU req.	MTC: x'_{14} = Communicants x'_{15} = Command acknowl. x'_{16} = PLA Command reliability x'_{17} = Timed out UU	MTC: x'_{20} = CLA command reliability x'_{21} = Communicants x'_{22} = Command acknowl.	MTC: x'_{23} = Secondary MU command reliability x'_{24} = Command acknowl.

INTERNAL I/O VARIABLES:

z_1 = Find PLA
z_2 = Delete PLA
z_3 = Find CLA
z_4 = Delete CLA
z_5 = Current PLA

Figure 2.2-4 (con't)

NETWORK MANAGER: Preliminary Control Map. Input/Output variables, Level β

OUTPUTS

to: ↓	from NC	from PC	from CC	from AMU
OI:	y'_1 = Command requests	MTC: y'_4 = Command requests y'_5 = Current PLAs OI: y'_6 = Current PLAs y'_7 = Forced PLA alert y'_8 = Zero-link alert y'_9 = Forced PLA display	MTC: y'_{13} = Current CLA's y'_{14} = Command requests OI: (y'_{13} = Current CLAs)	
RPT:	y'_2 = Handover-in successful	RPT: y'_{10} = UU cycle rate		RPT: y'_{16} = Secondary MU assignment complete
PT:	y'_3 = Unlocate track	PM: y'_{11} = Network occupancy (=network data?) y'_{12} = UU state	PT: y'_{15} = Unlocate track	

Figure 2.2-4 (con't)

NETWORK MANAGER: Preliminary Control Map. Input/Output variables, Level γ

ER (UU entry requests) I/O

INPUTS	OUTPUTS
$x"_1$ = New entry req.	$y"_1$ = UTC (USR) request
$x"_2$ = Re-entry req.	$y"_2$ = Handover-in successful
$x"_3$ = Handover-in req.	$y"_3$ = Tree allocation command
$x"_4$ = Acknowledgement (of $x"_3$)	$y"_4$ = Zero alert
$x"_5$ = Polling list	
$x"_6$ = Active entry req.	

CP (UU mode control processing) I/O

INPUTS	OUTPUTS
$x"_7$ = Passive mode req.	$y"_5$ = UTC (USR)
$x"_8$ = Reenter UU req.	$y"_6$ = PLA link deassignment command req.
$x"_9$ = Restart UU req.	$y"_7$ = CLA " " " "
$x"_{10}$ = Clear UU req.	$y"_8$ = Re-entry command
$x"_{11}$ = Re-initialize position tracking	$y"_9$ = Restart command
	$y"_{10}$ = Clear command
	$y"_{11}$ = CLA link deassignment command

TAC (Tree allocation change request) I/O

INPUTS	OUTPUTS
$x"_{12}$ = Tree allocation change request	$y"_{12}$ = Tree allocation commands

Figure 2.2-4 (con't)

into x'_{16}, x'_{20}, and x'_{23}, which are PLA (PORT Link Assignment) COMMAND RELIABILITY, CLA (CROSS Link Assignment) COMMAND RELIABILITY, and SECONDARY MU (Master Unit) COMMAND RELIABILITY at Level β.

Another question one wants to ask, based on the control map, is what functional relationship the module (subfunction) Adjacent MU Communications Assignment bears to the rest of the Network Manager, i.e., the Network Control subfunction, CLA Control, and PLA Control. Other than the common input/output COMMAND ACKNOWLEDGE and COMMAND REQUESTS, (which we take to be general terms, not, in fact, referring to the same data items), there seems to be no connection. In terms of HOS primitives, it is neither a Set Partition, a Class Partition, and certainly, not a Composition. Nor does it seem to be an abstract Control Structure derived from these primitives. Thus, one might question the validity of having it as part of the Network Manager function.

A more important problem is that the control map shows that in some cases the functions are not cleanly divided, i.e., that there is a lack of modularity. In plain language, there is confusion about which subfunction does what. For example if we look at Level β (the one which represents PPS Figure 3.4.2-1) on the control map, we see that the PLA Control generates the output y_8 = ZERO LINK ALERT (and similarly for other ALERTS). But if we look at Level γ which represents the verbal subdivision later in the Hughes document, we find ZERO LINK ALERT being generated by the UU Entry Request subfunction (this corresponds to the statement in PPS on page 3-38 under the description of the Network Control subfunction, "3.4.2.2.1.1 UU Entry requests....When a PLA is not found for an entering UU, a zero alert shall be generated for that UU"). The representation as a control map makes it immediately transparent that one of the descriptions must be incorrect, since two subfunctions in two distinct parts of the control map cannot generate the same identical data item.

While this may seem obvious on an intuitive level (i.e., that two distinct subfunctions cannot output the same item--it is either output by one or by the other), it is important to point out that (1) the control map, because it has several levels in the same representation, makes it easy to check for such inconsistencies, and (2) on a theoretical level, we

note that it violates one of the formal HOS axioms for the construction of control maps and AXES specifications, namely

> AXIOM 1: A given module controls the invocation of the set of functions on its immediate, and only its immediate lower level.

By extension, there is no way the PLA Control could invoke the UU Entry Request subfunction, since UU Entry Request is the immediate lower level of Network Control, not PLA Control.

Again, it should be pointed out that although in this simple case it seems obvious that the verbal description is in error, rather than the flow diagram, the control map provides a way for catching this immediately and a formal resolution for more complicated cases.

A further problem which is uncovered by the control map is that as one descends into more detailed levels of representation, inputs (or outputs) are introduced that didn't appear at all in the higher levels of representation. This is different from unclarity (vagueness or discrepancies) in representing a particular input item or items at different levels.

For example, some variables which appear at Level γ (like $x"_8$ = REENTER UU REQUEST) also appear at Level β (in this case as x'_{12} -- in a proper control map both should be designated by the same variable). However, seemingly analogous variables at Level γ, such as $x"_9$ = RESTART UU REQUEST, do not appear at Level β at all. This corresponds, again to the Hughes PPS Section 3.4.3.3.1.2.3 on pages 3-38, where we find "Upon receipt of a restart UU request..." analogously to Section 3.4.2.2.1.2.2, "Upon receipt of a reenter UU request..." However, in Figure 3.4.2.1 (the β Level), there is only the REENTER UU REQUEST being input from the MTC. Similarly, the HANDOVER-IN REQUEST appears on the γ Level and the β Level, but the CLEAR UNIT REQUEST only appears on the γ Level.

It should be pointed out that this sort of discrepancy, while, of course, violating HOS Axioms, is particularly unfortunate from the point of view of the Hughes PPS. Since it is in the β Level representation (Figure 3.4.2-1) that we are shown which modules external to the NM input various

data to subfunctions of the NM, if a new data input appears at the lower γ Level (the more detailed verbal descriptions in Section 3.4.2.2 ff), then we have no way of telling where (i.e., which subfunction) the data item comes from. In fact, for all one knows, the inputs could be output coming from either external or internal modules, since this is simply not stated, and one has to either guess as to the source of the data or search elsewhere in the documentation.

The advantages of a control map in this respect are that input variables of a subfunction which are external (i.e., come from a source outside the subfunction) MUST be carried through at all levels, so that in the complete control map, one can always trace the source. For example, if a is an input to the Network Control subfunction of the Network Manager, and that input comes from the MTC module, then in the complete PLRS control map one could trace the input as shown (Figure 2.2-5):

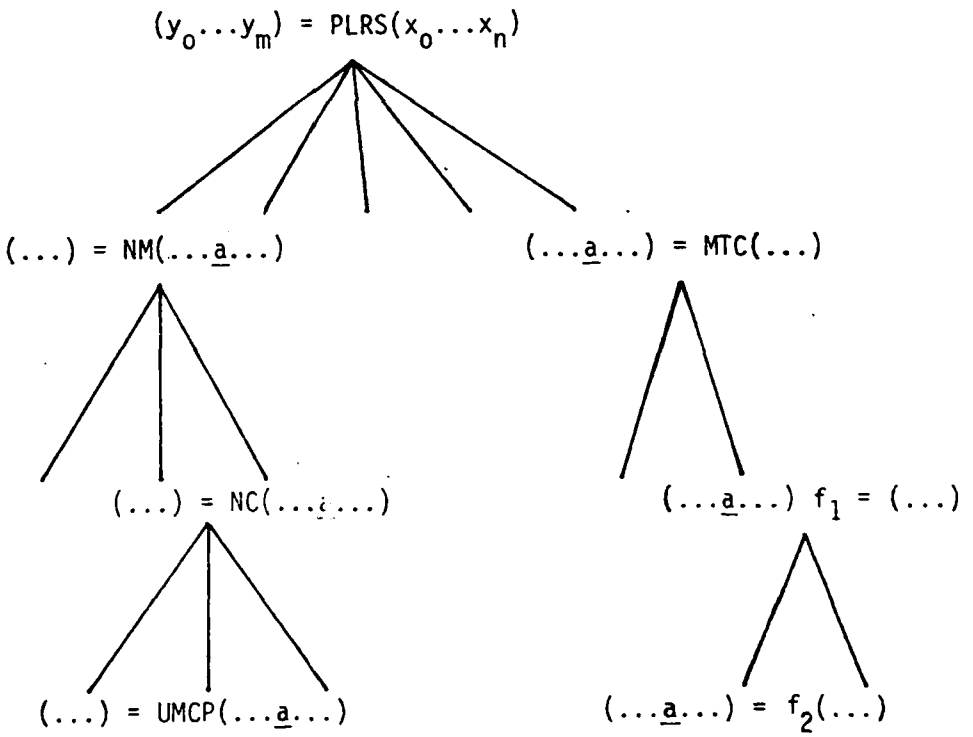


Figure 2.2-5

Finally, since input and output varibles must be carried through the control map as shown above, the preliminary Network Manager control map uncovers another problematic aspect of the Network Manager, as described in the PPS, namely calls to subroutines outside the module in question. These show up as the following structure (Figure 2.2-6):

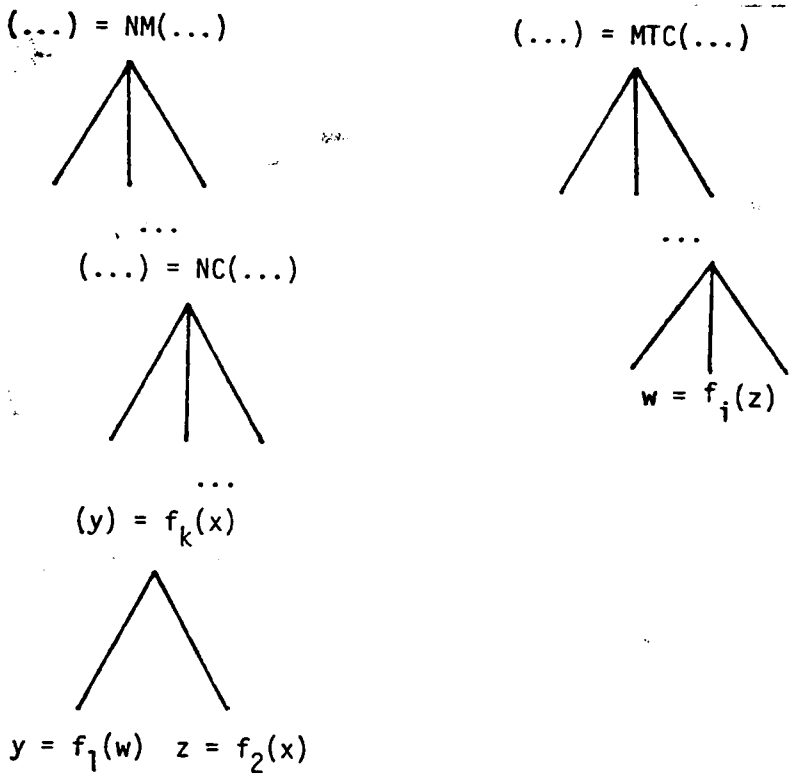


Figure 2.2-6

If we take w (= $x"_4$) = TREE ALLOCATION COMMAND, and z (= $y"_2$) = COMMAND ACKNOWLEDGEMENT, then according to Section 3.4.2.2.1.1.3 "Handover-in UUs" on page 3-38, which is a subfunction of Network Control: "...Network Control shall generate a tree allocation command containing the MU tree allocation to all handover-in UUs. Upon acknowledgement of this command, Network Control shall generate a handover-in successful notice." That is,

some subfunction, f_k, of the Network Control function computes (generates) y = HANDOVER-IN SUCCESSFUL NOTICE. However, we notice that part of f_k, namely the subfunction f_2 computes a value z, which is the input to an entirely different module (e.g., MTC), which in turn generates a value, w, which is the input to another subfunction, f_1, of the function f_k.

Not only do "hidden" calls to external functions destroy modularity, but in this case, there is no provision for error recovery should an acknowledgement fail to be generated. This is clearly shown by the preliminary control map, i.e., this would be invalid in a proper control map, since a function can control the access rights only to the inputs and outputs of its immediately dominated subfunctions, and here we have some function external to the Network Manager accessing an internal output of a subfunction of the Network Manager and vice versa.

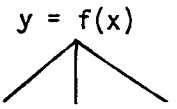
2.3 Understanding Data Properties

Once a preliminary survey of the system was made from the available documents, one of our first tasks was to identify the important data types. This turned out to be a particularly difficult task in this system, since there are many parameters (usually integers) used by the Network Manager which have only peripheral importance. What turned out to be the central and crucial data types for the Network Manager operation were quite complex and idiosyncratic objects. Most of our time was therefore spent on understanding the properties of these data types before we could specify operations employing them. Why this was so is discussed in detail in the next section.

3.0 THE PLRS DATA TYPES

3.1 The Network Manager

Turning now to our specification of the Network Manager (NM), what do we need to know in order to do the specification? Already at this first stage, the control map format provides a guideline as to the appropriate questions to ask first. We know that the top level of the control map will have the format:



That is, we must immediately begin thinking of the problem in terms of mathematical functions (mappings) acting on some input(s) to produce some output(s): f performs some action or computation on x to produce y. What are the x,y, and f of the Network Manager? A not unreasonable first assmption is that y = f(x) takes the form

$$Network_{NEW} = MANAGE(Network_{OLD}) \qquad (1)$$

That is, the Network Manager (NM) takes as input some state of the Network ($Network_{OLD}$) and performs some operations on it (the function MANAGE) to produce a new, reconfigured state of the network ($Network_{NEW}$). As it turns out, there are other inputs to the Network Manager other than just the current state of the Network: the NM must also know (1) the History of the network (e.g., the Unsatisfied Entry Count, which is a record for each User Unit (UU), of the number of other units which have tried to gain entry to the system through that UU and failed); and (2) Requests, for example, a request from the human Operator to force a particular assignment. There will, of course, also be certain secondary outputs regarding the state of the network and the result of the actions, such as ERROR messages, REQUESTS for more information to be sent to the Message Traffic Control (MTC), and the like. Thus the top-level function should really be something like:

$$(Network_{NEW}, Messages) = MANAGE(Network_{OLD}, History, Requests) \qquad (2)$$

These secondary inputs and outputs indicated in Equation (2) should not distract us from remembering that the basic function of the NM is to reconfigure the network, as indicated in Equation (1). Now in order to begin to describe this basic function, we have to set our priorities. Do we want to begin by looking at the actions (the function MANAGE) or on the object acted upon (the Network)? By casting the problem in this format, the HOS control map has already forced us to make a decision of this sort by highlighting the issue involved. In the case of PLRS, the answer is fairly straightforward: a preliminary survey of the problem indicated that the structure of the network was so extraordinarily complex, that we really needed to understand it thoroughly first, before we could expect to talk about the MANAGE function in any reasonable way.

3.2 Why Data Types?

What does it mean to understand the structure of the Network? We know, of course, that we will have to have some intuitive understanding of the Network as a working hypothesis. But can we at some point say that we have understood the Network and are ready tomove on to the next step? Here again, the HOS methodology provides an answer: we can begin describing the operations when we have identified the data types and have specified their behavior by a set of primitive operations and axioms. This defines the behavior of the data types so that we can go ahead with the control map (and specify other operations on those data types), because we know from the axioms what is permissible and what is not.

What makes the PLRS network so particularly complicated? (Why are its data types so difficult to specify?) There are several reasons:

1. The PLRS data types are unique to the PLRS system. In other systems, we may be working with such data types as files, vectors, scalars, lists, rational numbers, about which (a) people already have an intuitive understanding, and (b) there already exists a reference literature, both mathematical and computational, describing their properties. The objects which form the basis of the PLRS Network User Units (UUs) and Logical Times (LTs) are idiosyncratic objects, some of whose properties are determined by the particular implementation suggested in the Hughes documents. In a completely general specification, this might not be true.

2. The PLRS system is duplexed. This means that the mapping which relates UUs and LTs will not be a simple one-to-one correspondence.

3. The organization of Logical Time is itself complex. Because groups of time slots are interleaved with one another (even before scrambling), Logical Time is not a simple cyclic domain, but rather has a complex internal structure, which we must abstract away from in order to state what the simple operations relevant to the Network Manager are. (More on this below.)

One might want to ask (particularly in view of the complexity of the data-type specifications for PLRS), why should one care? Isn't there some easier way to specify the system?

In this case, the data-type specifications are particularly important for the reason given in Item (1) above: the data-types are unique to the system. We already know what to expect from vectors or rational numbers (for example, that one cannot divide by zero); about these idiosyncratic data types we have no idea what to expect in either of two cases:

(1) What operations are possible (e.g., have we overlooked a possible operation which would allow the program to run more efficiently);

(2) What operations are invalid (e.g., they yield no output or an incorrect output, causing a system error).

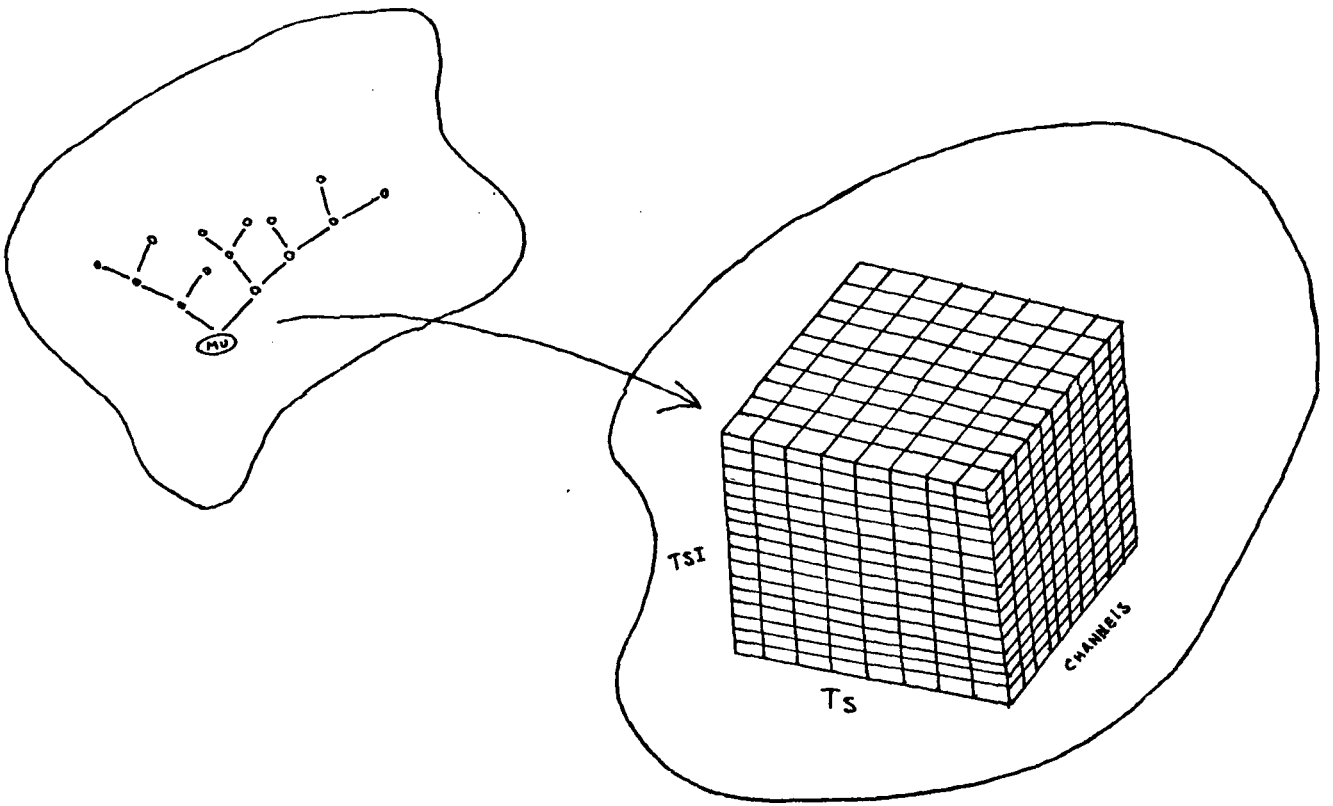
Trying to define the data types carefully in advance can help, for example, the programmer who subsequently uses them to know what can and cannot be done.

3.3 PORT Link Assignment

The basic concept of the Network is PORT Link Assignment, which is defined as follows: On the one hand there are User Units (radio-communication units), and on the other hand there are, for each User Unit, a set of time intervals in which that User Unit is allowed to operate. The mapping which assigns UUs to the set of time intervals in which they are to operate (transmit) is called PORT LINK ASSIGNMENT. The Hughes documents tend to be ambiguous in this respect, using the term "Port Link Assignment" (PLA) to mean both the process of mapping UUs onto time intervals, as well as the set of time intervals a UU is mapped onto. Hence, we find such apparently anomalous phrases as "unassigned PORT Link Assignment," which actually means a set of time intervals which has not yet been assigned to a UU by the PORT Link Assignment function[1]. In order to avoid confusion, in this report PORT Link Assignment will be used to refer only to the mapping, or assignment process, and the term Logical Time (LT) to represent the set of time intervals which is assigned to a particular UU.

Thus, it would seem our basic data types would be User Units and Logical Times, since the BASIC activity of the Network Manager is to assign a UU to one or more LT's: PORT Link Assignment. Of course, the Network Manager has other functions as well: the secondary assignments, known as CROSS LINK ASSIGNMENTS (for listening rather than transmitting) plus certain "housekeeping" or "clean-up" functions, which are needed to readjust the Network when a UU is reassigned. But we must not let this obscure the basic assignment function.

Now it turns out that the UUs are related to one another in certain ways, and the LTs are also related to each other in certain complex ways. A way of thinking about this which has proved helpful in defining the data types is to imagine a space, or universe, of UUs and another space, or universe, consisting of Logical Time intervals; PORT Link Assignment maps element in one space onto elements in the other (Figure 3.3-1).



MAPPING THE TREE (OR NETWORK) ONTO TIMESLOT SPACE

Figure 3.3-1

3.4 User Units

Let us first consider the User Units. Before we can state a formal algebraic definition of the data types User Units and Logical Time, we need to have some intuitive understanding of how they function, how they are related, and what operations on them are basic. In the case of the UUs, we know that each data UU represents an actual radio-communication device in the field. Messages from the Master Unit (MU) may be transmitted to some UUs directly, to others by being relayed along a chain of UUs. Similarly, inbound messages from some UUs are transmitted to the MU directly, others relayed via a chain of links between other UUs. These are the PORT Links, and together they form a PORT Path between a UU and and the MU. We can visualize this as a tree-like structure with the MU at the bottom, transmitting up the links to the UUs as shown in Figure 2-1. (Return messages are relayed back along the same paths.) If two User Units, U_i and U_j, are connected by a relay link, as shown in Figure 2-1, we will say that that U_i U-supports U_j, meaning that U_i transmits to U_j outbound from the MU, and receives from U_j inbound toward the MU. The notion of U-SUPPORT, then, will be the basic relation that UUs have with each other.

We notice that in this system (as described in the Hughes documents), U-SUPPORT as a relation between UUs has several properties, which are reflected in the axioms for data-type UU (Figure 3.4-1). First, we would like to say that U-support is transitive along the branches of a tree: if U_1 U-supports U_2, and U_2 U-supports U_3, then U_1 also U-supports (indirectly) U_3. This is stated in Axiom 2. Also, we would like to say that [illegible] cannot occur:

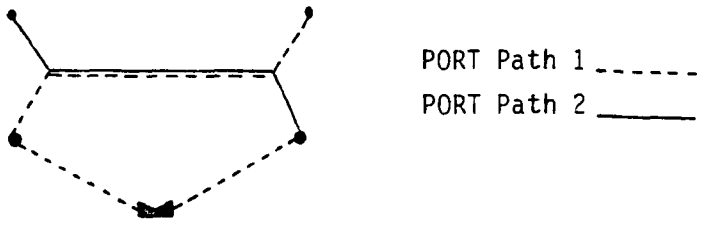


Figure 3.4-2
Forbidden Configuration

DATA TYPE: UU; /*User-unit*/;

PRIMITIVE OPERATIONS:

boolean = Usupp? (uu_1, uu_2);

tuple(of logical time) = PLA(uu);

UU = ALP(logical time);

AXIOMS:

WHERE u, u_1, u_2 ARE UU's;

WHERE M IS A CONSTANT UU;

WHERE t_1, t_2 ARE LOGICAL TIMES;

WHERE n IS A NATURAL;

(1) (Usupp?(u_1,u_2) ⊃ Not (Usupp?(u_2,u_1))) = True;

(2) ((Usupp?(u_1,u_2) & Usupp?(u_2,u_3)) ⊃ Usupp?(u_1,u_3)) = True;

(3) (Not(Eq(u,M)) ⊃ Usupp?(M,u)) = True;

(4) Not(Usupp?(u,u)) = True;

(5) ALP(Examine_Item(PLA(u),n)) = ^{1}u OTHERWISE $K_{Reject}(^2u)$;

PARTITION OF (u,n) IS

^{1}(u,n) | (1 ≤ n ≤ 2 & u≠M) ! (1 ≤ n ≤ 6 & u=M),

^{2}(u,n) | n=0 ! (n > 2 & u≠M) ! (n > 6 & u=M);

(6a) (Tsupp?(t_1,t_2) ⊃ Usupp?(ALP(t_1),ALP(t_2))) = True;

Figure 3.4-1

(6b) (Usupp?(u_1,u_2) = (OR(Tsupp?($ID_1^2(PLA(^1u_1))$, $ID_1^2(PLA(^1u_2))$),
(Tsupp?($ID_1^2(PLA(^1u_1))$, $ID_2^2(PLA(^1u_2))$),
(Tsupp?($ID_2^2(PLA(^1u_1))$, $ID_1^2(PLA(^1u_2))$),
(Tsupp?($ID_2^2(PLA(^1u_1))$, $ID_2^2(PLA(^1u_2))$),)

OTHERWISE (OR(Tsupp?($ID_1^6(PLA(^2u_1))$, $ID_1^2(PLA(^2u_2))$),
(Tsupp?($ID_2^6(PLA(^2u_1))$, $ID_1^2(PLA(^2u_2))$),
(Tsupp?($ID_3^6(PLA(^2u_1))$, $ID_1^2(PLA(^2u_2))$),
(Tsupp?($ID_4^6(PLA(^2u_1))$, $ID_1^2(PLA(^2u_2))$),
(Tsupp?($ID_5^6(PLA(^2u_1))$, $ID_1^2(PLA(^2u_2))$),
(Tsupp?($ID_6^6(PLA(^2u_1))$, $ID_1^2(PLA(^2u_2))$),
(Tsupp?($ID_1^6(PLA(^2u_1))$, $ID_2^2(PLA(^2u_2))$),
(Tsupp?($ID_2^6(PLA(^2u_1))$, $ID^2(PLA(^2u_2))$),
(Tsupp?($ID_3^6(PLA(^2u_1))$, $ID_2^2(PLA(^2u_2))$),
(Tsupp?($ID_4^6(PLA(^2u_1))$, $ID_2^2(PLA(^2u_2))$),
(Tsupp?($ID_5^6(PLA(^2u_1))$, $ID_2^2(PLA(^2u_2))$),
(Tsupp?($ID_6^6(PLA(^2u_1))$, $ID_2^2(PLA(^2u_2))$),

OTHERWISE $K_{Reject}(^3u_1, ^3u_2)$;

PARTITION OF (u_1,u_2) IS

$^1(u_1,u_2)$ $u_1 \neq M \& u_2 \neq M$,

$^2(u_1,u_2)$ $u_1 = M \& u_2 \neq M$,

$^3(u_1,u_2)$ $u_1 = M \& u_2 = M$;

END UU;

N.B. OR = n - place Logical Or

Figure 3.4-1 (con't)

That is, if U_1 U-supports U_2, then U_2 cannot U-support U_1. It should be noted that this property is not guaranteed a priori, but rather must be ensured by the way the search algorithm for finding PORT Link Assignments is constructed. We will therefore see, when we examine the control map, that this corresponds to one of the test modules in the program. We also note, and this is important, that although the Master Unit is clearly different in many respects from the User Units, it does behave like them in that it particupates in the U-SUPPORT relationship. Specifically, it U-supports all the UUs in one of its tree-like networks (Axiom 3). This is important for consistency in applying tests to pairs of UUs: if we ask (in Figure 2-1), "Does unit U_i support unit U_j?", we want the answer to be yes (TRUE); similarly, if we ask does the Master Unit M support unit U_i, we also want the answer to be yes. Thus the Master Unit behaves very much like zero among the natural numbers: zero is a number, the operation of addition and subtraction can validly be performed with it, and it can be the valid result of an operation; but it is also different, e.g., one cannot divide by zero.

In particular, it turns out, due to the way the PLRS system is structured, that one Master Unit can U-support several tree-like networks (at different times, or at different frequencies, of course), up to six; a UU can U-support units in no more than two trees (Figure 3.4-3).

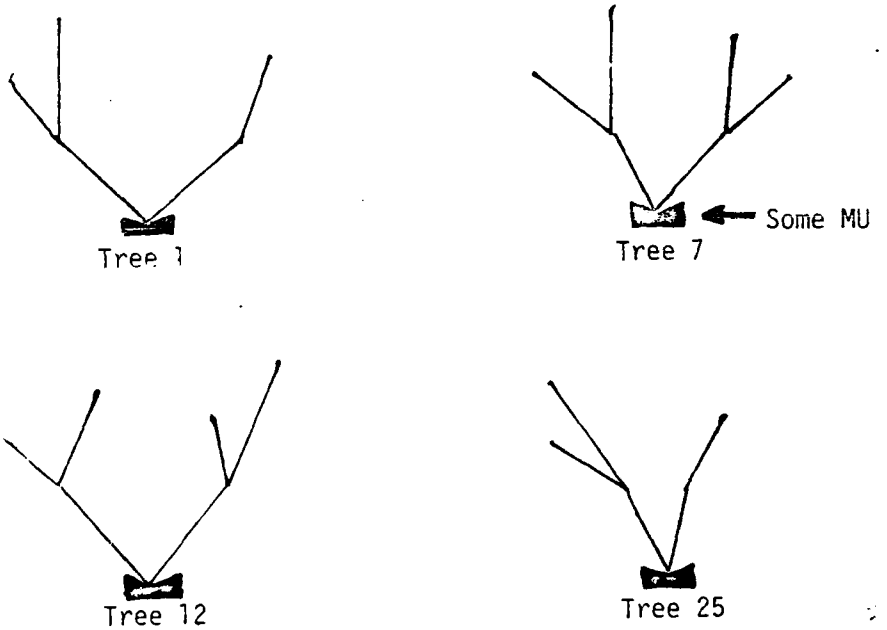


Figure 3.4-3

This is reflected in the way PORT Link Assignments (PLAs) are made: notice that UUs are assigned two Logical Times by the operation PLA, except for the MU, which is assigned six (e.g., $PLA(u_i) = (LT_{1_i}, LT_{2_i})$). But to see why this is related to U-SUPPORT, we need first to consider the other data type, Logical Time.

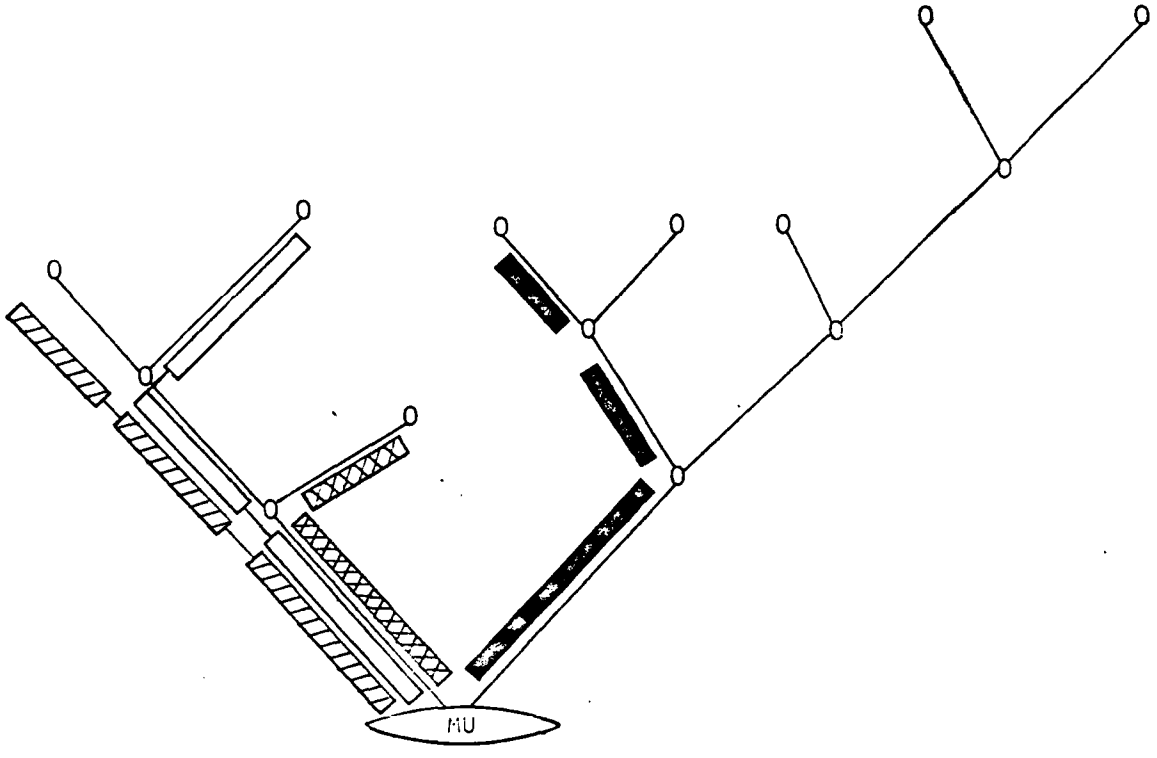
3.5 Logical Time

In the PLRS system, time is cyclic; transmissions are scheduled to occur over particular links at regular intervals, the rate depending on such things as the kind of unit involved; airborne units transmitting more frequently than manpack units, for example. Recall the tree-like network consisting of UUs, a MU, and the links between them (Figure 2-1). Now, each one of the links between two UUs is a radio-communication link, and thus occurs during some time interval. (For the sake of this example, we consider only outbound transmissions.)

Imagine these time intervals superimposed upon the link tree network as shown in Figure 3.5-1, where each color indicates the set of time intervals of one PORT path. Now Logical Time is organized into EPOCHS (64 seconds long) which are subdivided into 256 FRAMES, each of which consists of 128 smaller time intervals (Figure 3.5-2a). An appropriate image of what happens is to imagine the time intervals in Figure 3.5-1 as colored neon lights, and there is a flash up and back one of these PORT paths in each of the 256 frames of an epoch. Clearly, many of them will flash more than once and at regular intervals. Since there are, however, 128 time intervals in each frame, and (as we shall see later) since PORT paths are limited to no more than four levels, the Logical Time intervals corresponding to PORT Paths in several different trees (networks) could flash during one Frame.

Now imagine a string tied to the bottom on each of the sets of time intervals corresponding to one of the PORT paths in the tree-network of Figure 3.5-1, and both ends pulled apart so that all of the columns of time intervals stood up vertically from a base (as in Figure 3.5-3). We could then also, as it were, tip all of the columns of time intervals over on their sides so they would lie end-to end--this would then give

REPRESENTATIVE TIME SLOTS FOR THE NETWORK IN FIGURE



Time intervals of PORT Path 1:

Time intervals of PORT Path 2:

Time intervals of PORT Path 3:

Time intervals of PORT Path 4:

Figure 3.5-1

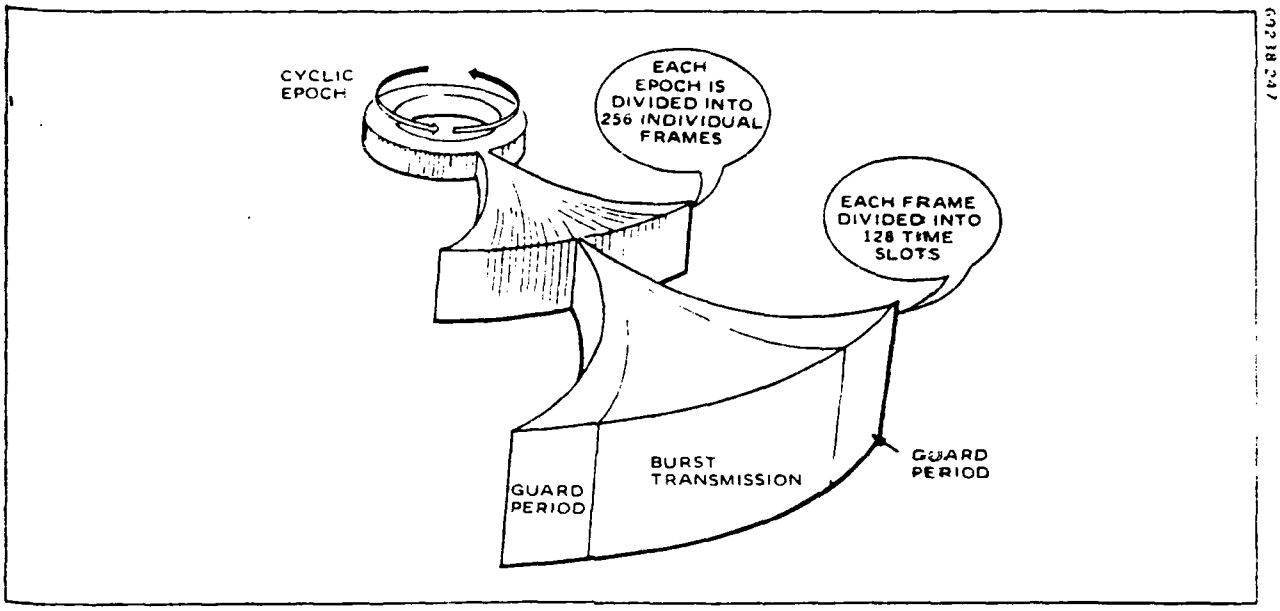


Figure A. The Cyclic Timing Structure of a PLRS Network is Divided into Epochs, Frames, and Time Slots. Only one community member at a time can transmit a burst during any given time slot. Most actions that a User Unit can be programmed to do are repeated each epoch.

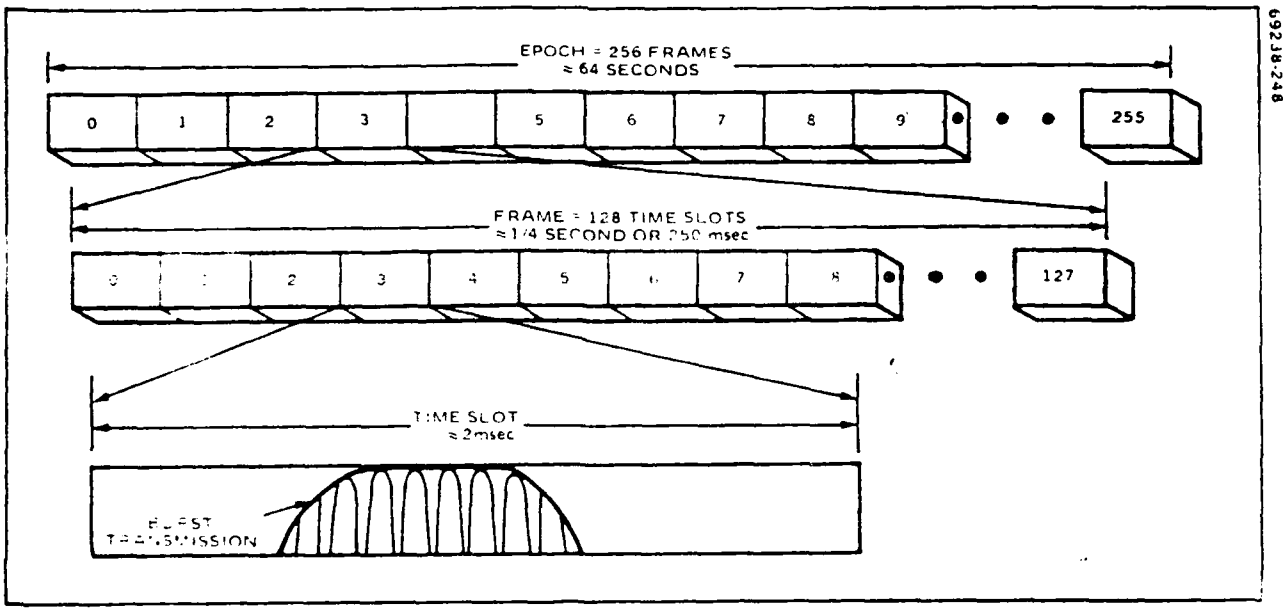


Figure B. The Time Divisions of the PLRS EDM. The capacity of the PLRS network is based on 128 time slots within each of 256 frames (total of 32,768 time slots per epoch).

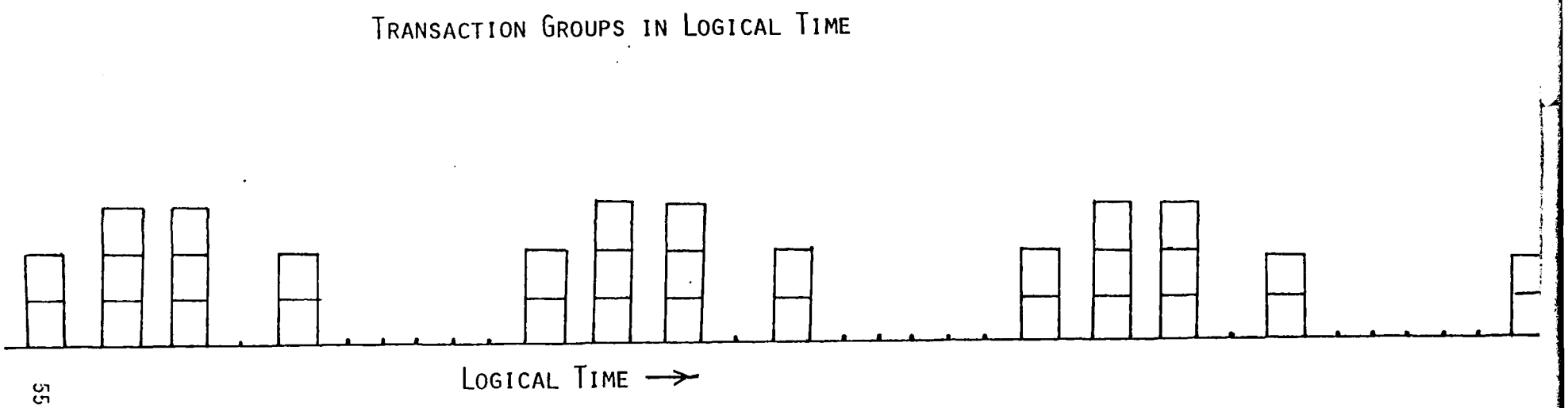


Figure 3.5-3

us a linear sequence of time intervals (Figure 3.5-4, upper left-hand corner) which is necessary, because time is, in fact, linear.

There are two additional factors which complicate this rather straightforward interpretation of logical time (as represented in Figures 3.5-1, 3.5-2, 3.5-3). Both can be ignored, for all practical purposes, by the Network Manager but need to be mentioned briefly, so as to avoid confusion later. Each of the columns in Figure 3.5-3 represent only outbound transmissions. Since, in reality, transmissions are both inbound and outbound, there must be more than four time intervals involved. In fact, there are 16, structured as shown in Figure 3.5-5. While there is only one outbound transmission by each UU along a PORT Path, there are several inbound transmissions, and in addition, one time interval (the fifth) is reserved for special purposes, such as requesting entry into the network. One set of 16 time intervals structured in this way is called a TRANSACTION GROUP, and the time intervals in one TRANSACTION GROUP are referred to as TIME SLOT INDICES, e.g., (Figure 3.5-5), the MU transmits in Time Slot Index (TSI) #0, the first inbound transmission is in TSI #6, etc. Since no Unit is allowed to be more than four links (levels) away from the MU in order to specify where in a TRANSACTION GROUP a Unit operates, we need to specify only the level of the Unit's associated time intervals.

Since all transaction groups have the same pattern of transmissions, given the level, we know exactly which time intervals within a transaction group a unit uses for transmitting and which for receiving. Hence the Network Manager can ignore the internal structure of the transaction groups and need only know the (1) the level of the time intervals to which a unit is assigned, and (2) which transaction group they are in.

How is the latter specified? This involves the second aspect of the organization of time which the Network Manager can ignore, because although complex, it is fixed. Logical Time is mapped into real-world time in a complex way. If we take all the time intervals in a frame (128) and arrange them in a rectangle (or a box, if we have operations at different frequencies) so that the transaction groups are vertical columns (Figure 3.5-5), then we can unwind, or scramble, these Logical Time intervals into real time in some random way for security purposes. This latter scrambling can be ignored entirely by the Network Manager.

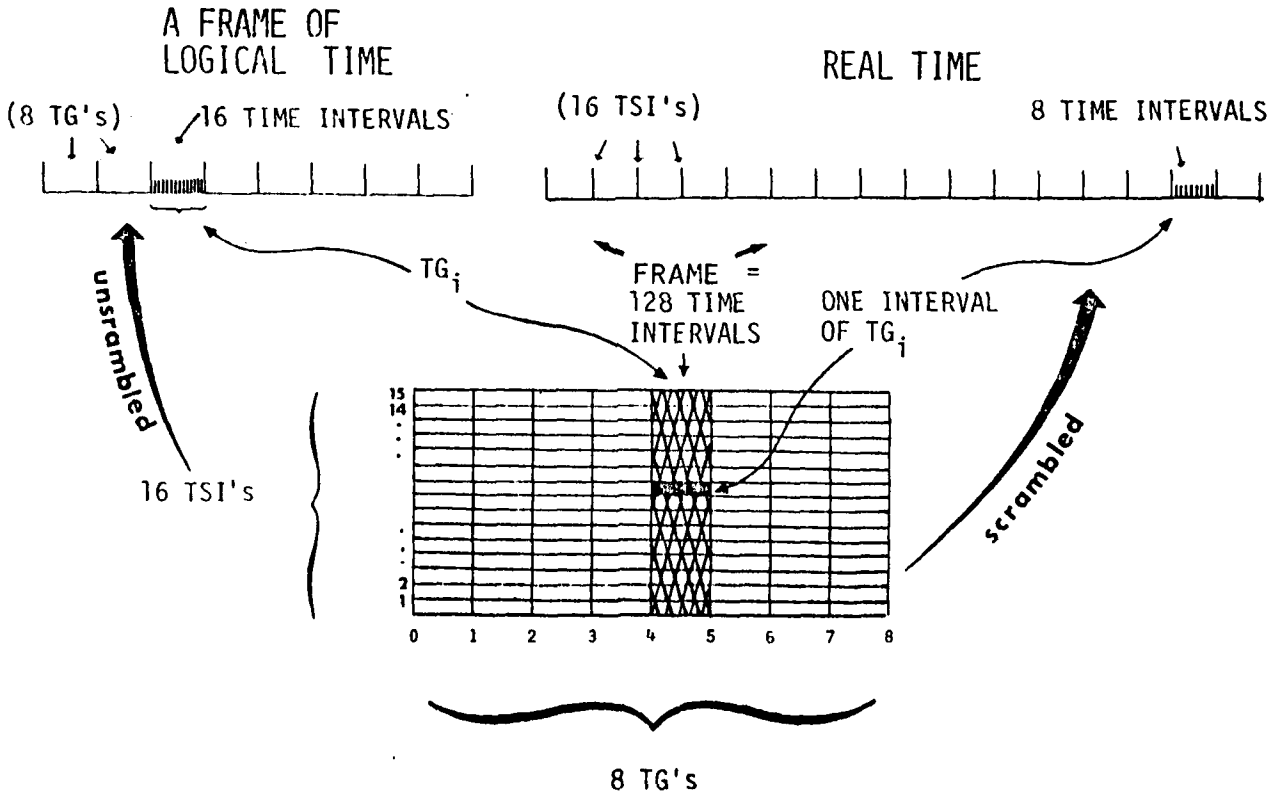
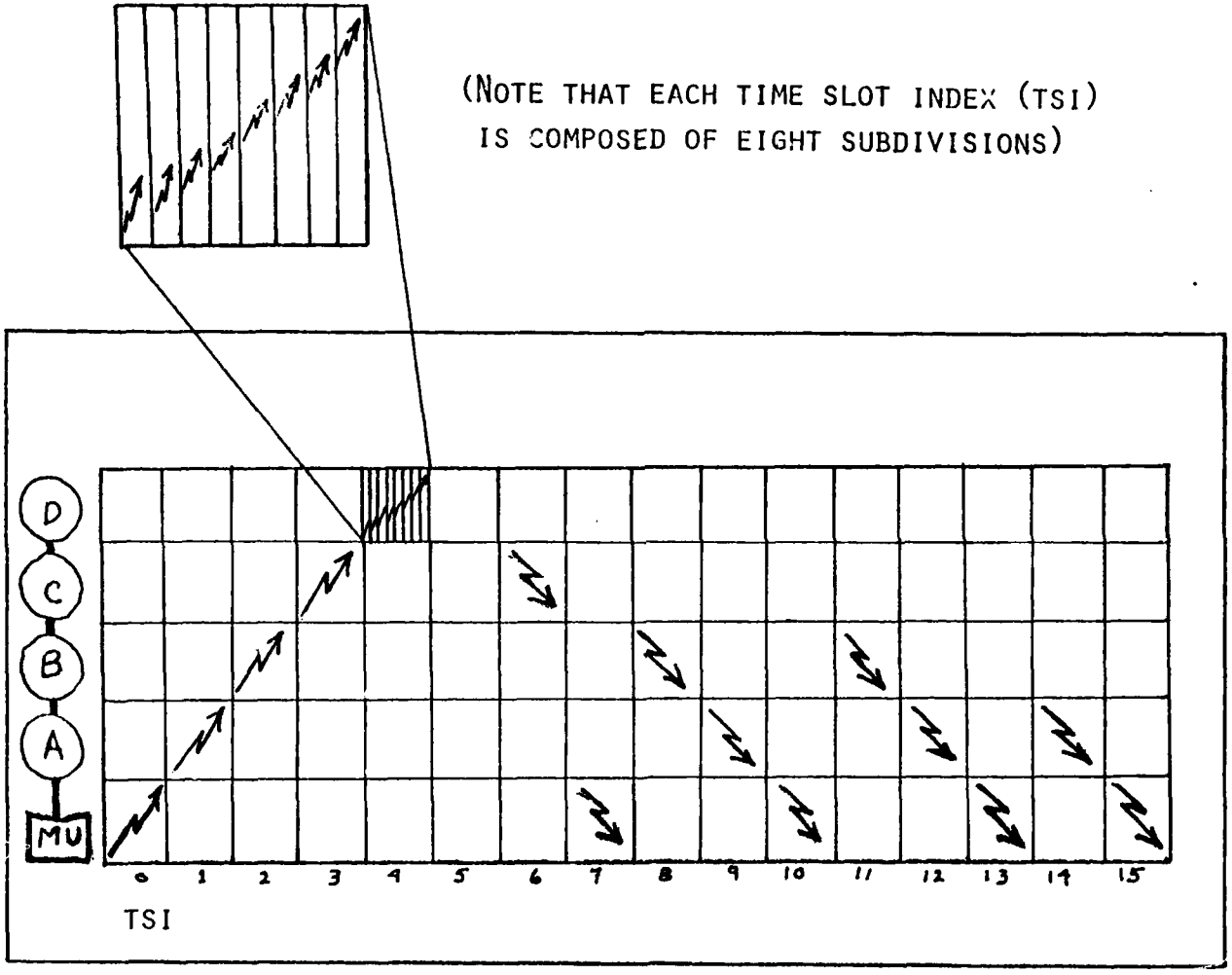

A FRAME OF
LOGICAL TIME
(8 TG's)
16 TIME INTERVALS
REAL TIME
(16 TSI's)
8 TIME INTERVALS
TG$_i$
FRAME =
128 TIME
INTERVALS
ONE INTERVAL
OF TG$_i$
unscrambled
scrambled
16 TSI's
15
14
2
1
0 1 2 3 4 5 6 7 8
8 TG's

Figure 3.5-4



TIME SLOT INDICES COMPRISING A TRANSACTION GROUP

Figure 3.5-5

In the representation of a Frame as a box or rectangle, however, the transaction groups are the columns, and as such can be assigned numbers. Now we will make it a requirement (as does Hughes) of our system that in each of the 256 frame "boxes," or "rectangles," that a column with the same number represents a transaction group (PORT Path) from the same tree (network). If we now imagine the 256 Frame boxes all lined up linearly in an epoch, and consider, say, column #35 in each box, a neon flash in that column will correspond to a neon flash up and down some (different) PORT Path in the same tree (network). So now if we know the level and tree number (i.e., which column number within a frame/box), all that remains to specify when a UU operates is to say which frames it operates in. But since the transmissions along the PORT Paths are cyclic (e.g., every 4 frames), all we need to specify is the Period (Per) and the Start Frame (SF).

Thus these four numbers completely and exhaustively specify which Logical Time intervals a UU operates in: Period and Start Frame are sufficient to tell which frames a unit's transaction group (PORT Path) is assigned to operate in, the Tree Number (TN) tells which column of time intervals (Transaction Group) in each frame, and Level (Lev), as explained above, determines which time intervals (TSI's) within a column of time intervals (Transaction Group).

Hence we can think of a UU's PORT Link Assignments as assignments to operate at a particular Logical Time, specified by these four integers (c.f., Figure 3.5-6):

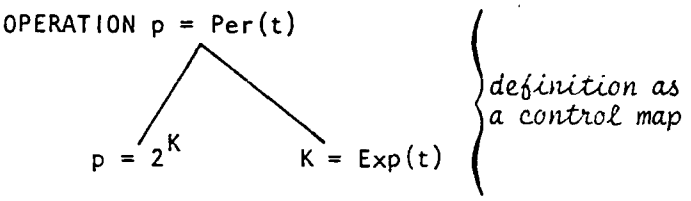
$$t = (\text{Exp.}, \text{Lev}, \text{TN}, \text{SF}) \qquad (3)$$

Since the data format in the documentation [14] only allows 3 bits to specify period, it must be the exponent (i.e., $2^{exp} = Per$), not the period itself (Figure 3.5-7) that is used to specify a Logical Time. This fact is reflected in the specification of our data-type Logical Time, as indicated in Figure 3.5-8[2]. We can then define Per(t) as a non-primitive operation.

PORT LINK ASSIGNMENT COMMAND, STD P. 3.3-13

60		P	L	TN	SF	
CMD TYPE	, ⟨PERIOD⟩	, ⟨LEVEL⟩	, ⟨TREE NUMBER⟩	, ⟨START FRAME⟩		
5	3	2	6	8		(BITS)

Figure 3.5-6




```
OPERATION:  p = Per(t);
   WHERE p,t ARE NATURALS;
p = 2^K  JOIN K = Exp(t)
END Per;
```

Figure 3.5-7

DATA TYPE: Logical time (of M);

PRIMITIVE OPERATIONS:

integer = Exp(t);
integer = Lev(t);
integer = TN(t);
integer = SF(t);
boolean = Teq?(t_1,t_2);

AXIOMS:

WHERE T_M IS A CONSTANT SET OF Logical times;

WHERE t_M IS A T_M;

WHERE t, t_1, t_2 ARE Logical times AND ARE NOT T_M;

[1] $0 \leq Exp(t) \leq 7;$

[2] $-1 \leq Lev(t) \leq 3;$

[3] $0 \leq TN(t) \leq 63;$

[4] $0 \leq SF(t) \leq 255;$

[5] $(Exp(t_1)+1 = Exp(t_2)) = (SF(t_2) - SF(t_1) = 2^{Exp(t_1)-1};$

[6] $Exp(t_M) = 0;$

[7] $0 \leq TN(t_M) < 64;$

[8] $Lev(t_M) = -1;$

[9] $SF(t_M) = 0;$

[10] $Teq?(t_1,t_2) = AND(Exp(t_1) = Exp(t_2),$
$Lev(t_1) = Lev(t_2),$
$TN(t_1) = TN(t_2),$
$SF(t_1) = SF(t_2) \bmod Per(t_1)):$

END Logical time;

Figure 3.5-8

More importantly, now, we need ask, what properties do Logical Times have which will be useful to us in specifying the system? We remember that there was a rough intuitive notion of one UU supporting another UU, which we tried to formalize (by axiomatizing). When one UU supports another along a PORT Path, however, the time slots which correspond to the links also stand in a particular relation to one another, namely they follow one another successively in a transaction group of Logical Time intervals. But this is a strictly numerical relationship, and easy to check automatically; this, this should be our basic relationship, and support between UUs dependent upon whether their assigned time slots are adjacent. To make this clear, we have separated the two meanings of "SUPPORT": U-support (between UUs) and T-support (between Logical Times).

This distinction is extremely important, because the two behave quite differently. The Logical Time intervals always "exist" and stand in the same relation to one another regardless of whether or not some UU is designated to operate in them. Thus if t_1 and t_2 stand in the relationship such that t_1 T-supports t_2, we will say Tsupp?(t_1,t_2) = TRUE regardless of whether or not they have UUs assigned to them by the PORT Link Assignment function (PLA). This can be determined solely from the primitive operations on t's, such as Exp(t), etc., as shown in the definition of the operation Tsupp? in the data-type specification (Figure 3.5-9).

3.6 U-support vs T-support

We are now in a position to say explicitly how to answer the question, "does U_1 U-support U_2?" computationally. Usupp?(u_1,u_2) = TRUE, if and only if one of the t's assigned by PLA to u_1 happens to T-support one of the t's assigned to PLA to u_2. That is, suppose for a moment that u_1 and u_2 have only one PORT Link Assignment each instead of two: t_a = PLA(u_1) and t_b = PLA(u_2). Then if Tsupp?(t_a,t_b) = TRUE, we know that Usupp?(u_1,u_2) = TRUE as well. Since Tsupp? is defined by computable functions, we would know exactly how we can answer the question Usupp? in our program should it arise.

Unfortunately, because of the required duplexing, things are not this simple. For many good operational reasons[3], it is necessary for each

OPERATION: boolean = Tsupp? (t_1, t_2)

WHERE t_1, t_2 ARE Logical Times;

Tsupp? (t_1, t_2) = ($Exp(t_1) \geq Exp(t_2)$) &
($Lev(t_1) < Lev(t_2)$) &
($TN(t_1) = TN(t_2)$) &
($SF(t_1) \equiv SF(t_2)$ *mod* $Per(t_1)$) ;

END Tsupp? ;

Figure 3.5-9

UU to have not one, but two PORT Link Assignments; that is, PLA(u) = (t_a,t_b), a duple. (Leaving aside the behavior of the MU, which we consider later.) So while the inverse of the PLA mapping (ALP) is single valued (ALP(t) = u; there is only one UU assigned to a Logical Time), PLA itself is two-valued, and the question "what is the PLA of U_i?" is meaningless, since there are two, and we must at all times know which one we are referring to. (We will see later how this causes problems when we look at some of the proposed tests in the search algorithm in the Control Map.)

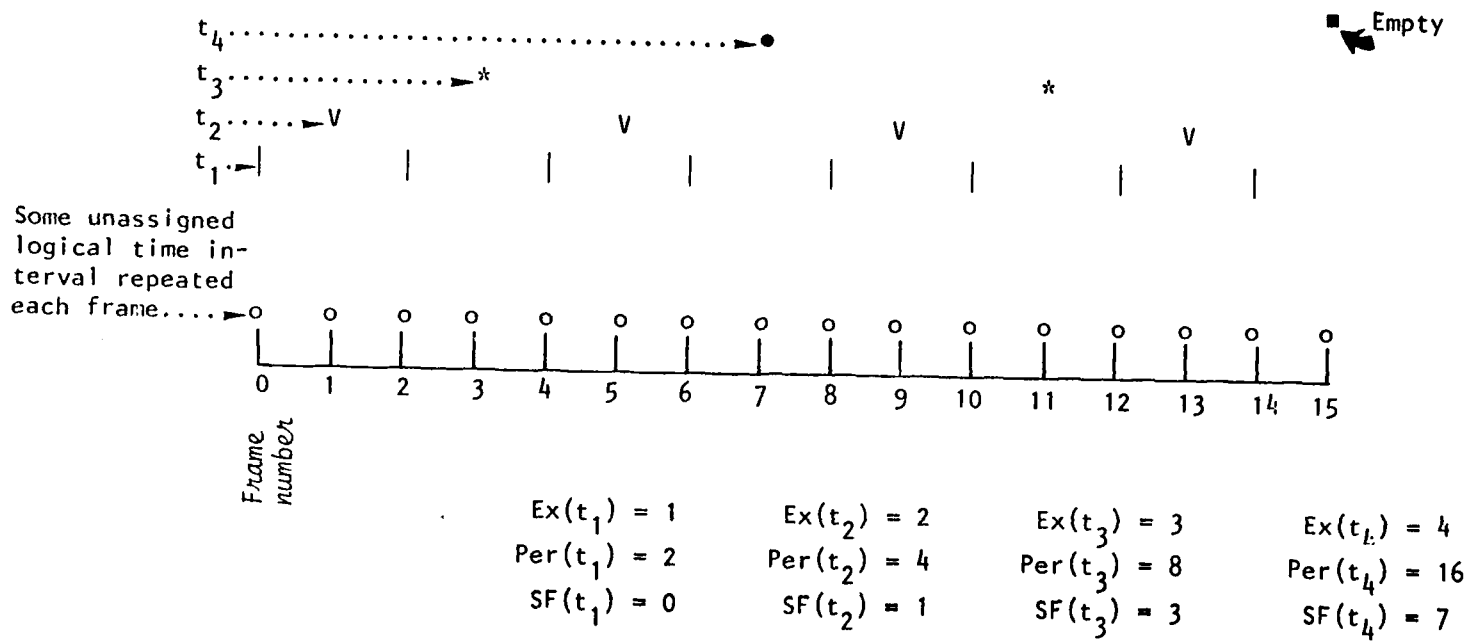
This affects our calculation of Usupp? If $PLA(u_i)$ equals (t_a,t_b) and $PLA(u_2)$ equals (t_c,t_d), then $Usupp?(u_1,u_2)$ is true, if either $Tsupp?(t_a,t_c)$ or $Tsupp?(t_a,t_d)$ or $Tsupp?(t_b,t_c)$ or $Tsupp?(t_b,t_d)$ are true!

In other words, if we want to ask, as part of some test in the Network Manager module, if one UU supports another, we may have to try all four possibilities (Figure 3.4-1, Axiom 6b). Note the difference in complexity between Axiom 6b, which is stated in terms of PLA, and Axiom 6a, which is stated in terms of the inverse function, ALP[4]. Because of this we will always want to try to state our algorithms, if possible, in terms of the Logical Time relations, rather than UU relations, since UU relations will have to be translated back into Logical Time relations to be calculated anyway, and the translation is anything but straightforward. We will examine this more carefully when discussing the Control Map for the PLA Control submodule.

3.7 Logical Time Axiom #5

Returning briefly to the specification for the data type Logical Time, there are two points which deserve further comment. First is the rather strange and complex looking Axiom 5 (Figure 3.5-8). What this says is that there is no optionality in dividing up unassigned (available) Logical Times, but rather that they are structured according to a particular pattern. For example, suppose in Figure 3.7-1[5] that all the time intervals marked by little circles were available for assignment(i.e., not assigned to a UU: ALP(t) = REJECT). Then there is no a priori logical

SAMPLE 16-Frame Epoch:



$Ex(t_1) = 1$	$Ex(t_2) = 2$	$Ex(t_3) = 3$	$Ex(t_4) = 4$
$Per(t_1) = 2$	$Per(t_2) = 4$	$Per(t_3) = 8$	$Per(t_4) = 16$
$SF(t_1) = 0$	$SF(t_2) = 1$	$SF(t_3) = 3$	$SF(t_4) = 7$

PRIMITIVE AXIOM: $Exp(t_1) + 1 = Exp(t_2) \leftrightarrow SF(t_2) - SF(t_1) = 2^{Exp(t_1)-1}$

Figure 3.7-1

reason why we could not divide them up as follows: as <u>two</u> possible Logical Times available for PORT Link Assignments:

	Per	Lev	TN	SF	
t_1:	(2	x	y	0)	(level and tree given as x and y since they are not relevant to this discussion)
t_2:	(2	x	y	1)	

Alternatively, another logically possible division would be to have <u>four</u> possible logical times with periods of 4:

t_1:	(4	x	y	0)
t_2:	(4	x	y	1)
t_3:	(4	x	y	2)
t_4:	(4	x	y	3)

Or three logical times for possible PORT Link Assignments with different periods:

t_1:	(2	x	y	0)
t_2:	(4	x	y	1)
t_3:	(4	x	y	3)

There are, of course, many, many more possibilities, even for this 16-frame epoch, and enormously more for a real 256-frame epoch. However, as far as we can determine (c.f. [14] pp. 3.5-18/19), the actual PLRS program divides them up by a standard algorithm: begin with the logical time having the lowest possible start frame and lowest possible period, call it t_1; take the next lowest start frame and next lowest possible period among the remaining available logical times (Lev and TN being held constant), and call that t_2; from the remaining logical times choose the next lowest start frame and next lowest period for t_3, and so on. Thus from the unassigned logical times, we always get the same "mix" of possible PORT Link Assignments, as represented by the t_1, t_2, t_3, t_4 actually shown in Figure 3.7-1.

It should be remarked, incidentally, that we wondered whether this could possibly have been left as a user option: to vary the mix of logical times (possible PORT Link Assignments) at different rates (periods), depending on the mix (ratio) of airborne, manpack, and other units in a division. Obviously if one had all fast rate (low period) logical times available, one would have less total possible PORT Link Assignments, or if one had the other extreme, all slow rate (high period) logical times there would be many more possible PORT Link Assignments. One would not use either extreme, just as one does not turn the shower on all the way hot or all the way cold, it is helpful to be able to adjust the temperature.

3.8 Other Operations on Logical Times

Having our basic specifications of the data types UU and Logical Time in hand, it is then simple to define additional operations on those data types as we need them for the program. For example, consider some additional operations on Logical Time. The operation Tsupp? (Figure 3.5-9) tells us if, for any two Logical Times, one supports the other. We may want to know if one directly supports the other--e.g., in Figure 3.8-1, t_1 supports t_3, but it does not directly support t_3. The time interval t_2, on the other hand, does directly support t_3. It is a simple matter to write an operation which tests this, as shown in the definition of DTsupp? in Figure 3.8-2.

Similarly, we find in doing the control map that we want to ask, for a particular Logical Time, which Logical Time it is supported by. This can also be defined from previous operations, as shown in Figure 3.8-3, Derived Operation DTsupp. Note the difference in these two operations: the first asks, given two times, does one support the other; the other calculates which time supports the one already given:

$$\text{boolean} = \text{DTsupp?}(t_1, t_2) \tag{5}$$

$$t_1 = \text{DTsupp}(t_2) \tag{6}$$

We will see, when we start writing the specification for a section of the Network Manager that we actually do need both different operations.

REPRESENTATIVE TIME SLOTS FOR THE NETWORK

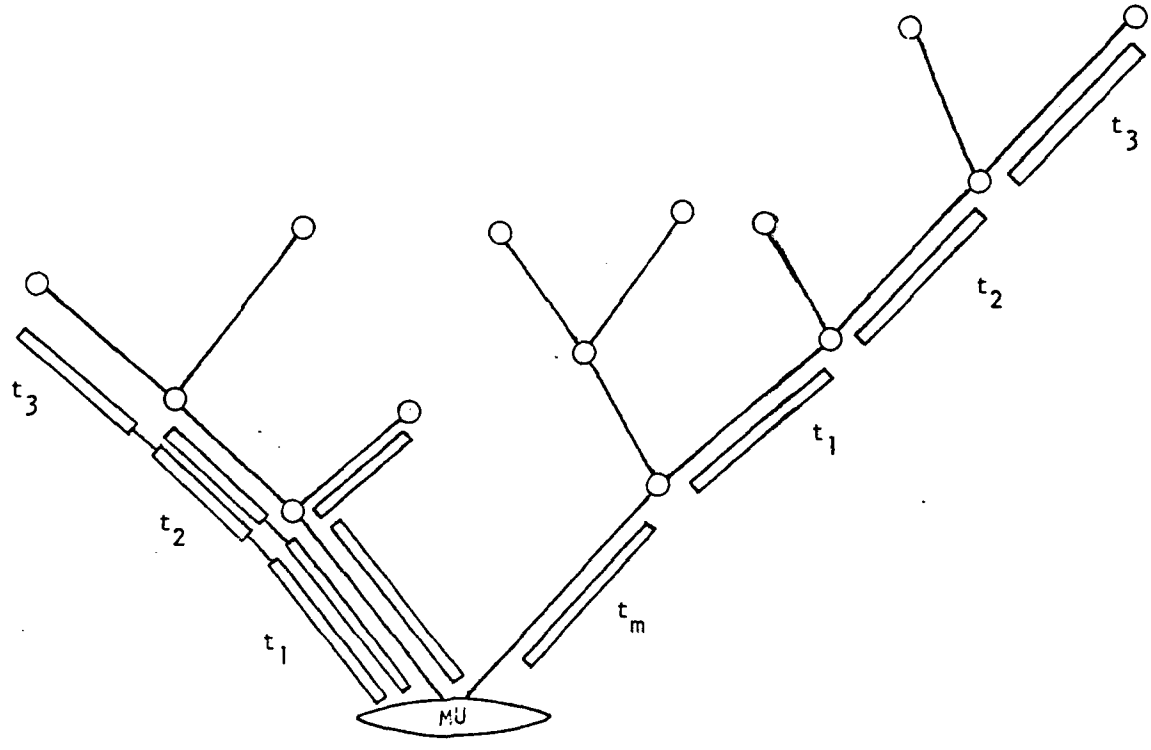


Figure 3.8-1

OPERATION: boolean = DTsupp?(t_1,t_2);

WHERE t_1,t_2 ARE Logical times;

DTsupp?(t_1,t_2) = Tsupp?(t_1,t_2) AND (Lev(t_1)+1 = Lev(t_2));

END DTsupp?

Figure 3.8-2

DERIVED OPERATION: t_1 = DTsupp(t_2);

WHERE t_1,t_2 ARE t's;

DTsupp?(t_1,t_2) = (t_1 = DTsupp(t_2));

END DTsupp;

Figure 3.8-3

There are several other operations defined with the data-type specifications (Figure 3.8-4). Some of these are referred to as needed in discussing the control map for FIND-PLA (Section 4).

3.9 Other Data Structures in the Network Manager

It should be noted that in addition to the data types Logical Time and User Unit that were specified for use in the following control map, there will need to be other data types and structures specified as more modules of the Network Manager are completed. For example, in order to specify the assignment and clean-up modules of PLA Control as well as the various modules making up the preliminary part of PLA Control, we would need the data structure for PORT Link Assignment itself. Intuitively, this could consist of a tree-like structure, as in Figure 3.9-1:

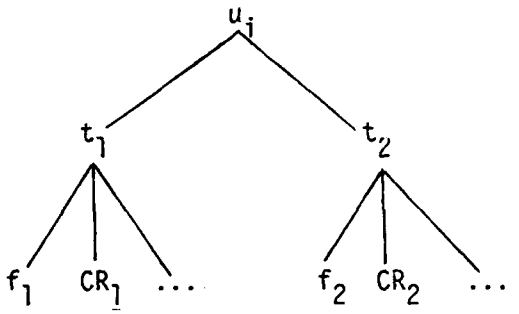


Figure 3.9-1

where u_i is some User Unit, t_1 and t_2 are the Logical Times such that (t_1,t_2) = PLA(u_i) and f, CR, ... are various parameters associated with the particular PORT Link Assignment (u_i,u_j), such as whether or not it is FORCED, it's COMMAND RELIABILITY[6], etc. Such a data structure would have primitive operations, such as:

SAMPLE PRIMITIVE OPERATIONS FOR PLA:

port-link-assignment = Assign(user-unit, logical-time, integer)
port-link-assignment$_1$ = Deassign(port-link-assignment$_2$,integer)
user-unit = UU(port-link-assignment)
logical-time = LT(port-link-assignment, integer)

Definition of ACTIVE IN SAME FRAME:

OPERATION: boolean = AISF(t_1,t_2);

WHERE t_1,t_2 ARE Logical times;

ASIF(t_1,t_2) = OR(SF(t_1) = SF(t_2) mod Per(t_1),

SF(t_1) = SF(t_2) mod Per(t_2));

END ASIF;

Alternative formulation:

AISF(t_1,t_2) = (SF(t_1) = SF(t_2) mod MIN(Per(t_1),Per(t_2)))

Figure 3.8-4

and axioms such as:

SAME AXIOMS FOR PLA:
WHERE u IS A User Unit,
(t_1,t_2) ARE Logical Times,
k IS A Natural,
pla IS A PORT Link Assignment;

(1) t = LT(Assign(u,t,k)),k);
(2) u = UU(Assign(u,t,k));
(3) PLA(u) = (LT(Assign(u,t_1,0),0, LT(Assign(u,t_2,1)1));
(4) ALP(t) = UU(Assign(u,t,k));
(5) REJECT = LT(Deassign(pla,k),k);
(6) u = UU(Deassign(pla,k),k);

Clearly, additional axioms would be needed, depending upon design considerations. For example, if we wanted the assignment module, which follows the FIND-PLA module, to simply replace a Logical Time in a PORT Link Assignment, then we might leave the assignment axioms as they are, since Axiom (1) already does this; if we were to insist (for whatever reasons) the desassignment was required before reassigning a new Logical Time, the we might add Axioms (7) and (8):

(7) pla = Assign(u,t_1,k);
(8) REJECT = Assign(u,t_2,k);

In any case, it would be to this data structure that we could then add additional operations and accompanying axioms to deal with such parameters as COMMAND RELIABILITY and FORCED-PLA. For example, suppose we wanted to indicate whether or not a PLA was forced. We could then add the primitive operations:

SAMPLE PRIMITIVE OPERATIONS FOR FORCED-PLA

Rforce(port-link-assignment, integer) = port-link assignment;
Rforced?(port-link-assignment, integer) = boolean
Unforce(port-link-assignment, integer) = port-link-assignment;

and accompany them with axioms, such as:

SAMPLE AXIOMS FOR PRIMITIVE OPERTIONS FORCED-PLA

```
Rforced?(Rforce(pla,k),k) = True;
Rforced?(Unforce(pla,k),k) = False;
```

Note that we need each time we refer to a unit's PORT Link Assignment structure, we need to include the integer which indicates whether it is the first or second assignment (pairing of the u and some t) that is being referred to. This issue is also discussed in Section 4.2.3.3 in this report. We could get around this by defining our data type Logical Time differently; that is, adding the concept of assignment of parameters directly to the data type Logical Time:

SAMPLE ADDED PRIMITIVE OPERATIONS FOR DATA TYPE LOGICAL TIME:

```
logical-time = Assign(user-unit, logical-time);
logical-time = Deassign(logical-time);
logical-time = Rforce(logical-time);
logical-time = Unforce(logical-time);
boolean = Rforced?(logical-time);
```

with the corresponding axioms:

SAMPLE ADDED AXIOMS FOR ABOVE PRIMTIIVE OPERATIONS:

```
WHERE u IS A User Unit,
      t IS A Logical Time;
u = ALP(Assign(u,t));
REJECT = ALP(Deassign(t));
Rforced?(Rforce(t)) = True;
Rforced?(Unforce(t)) = False;
```

While this is clearly more elegant mathematically[7] than the separate data structure for PORT Link Assignment, it leaves us with the problem, discussed elsewhere in this report (Section 3.6) of not being able to reference, or address, a <u>particular</u> PORT Link Assignment of a UU, other than to give the specification of the Logical Time involved. That is, as we shall see in Section 4, the FIND-PLA control map, tests are continually being proposed which refer to UUs, and there is no way (given

this latter specification for Logical Time) to refer to a UU's PORT Link Assignment, since we still have PLA(u) = (t_1,t_2) and so there are two. The former solution, to have a separate data structure for PORT-Link-Assignment, including a reference integer, is considerably more clumsy. In view of the necessity for duplexing, this is a problem which deserves more study.

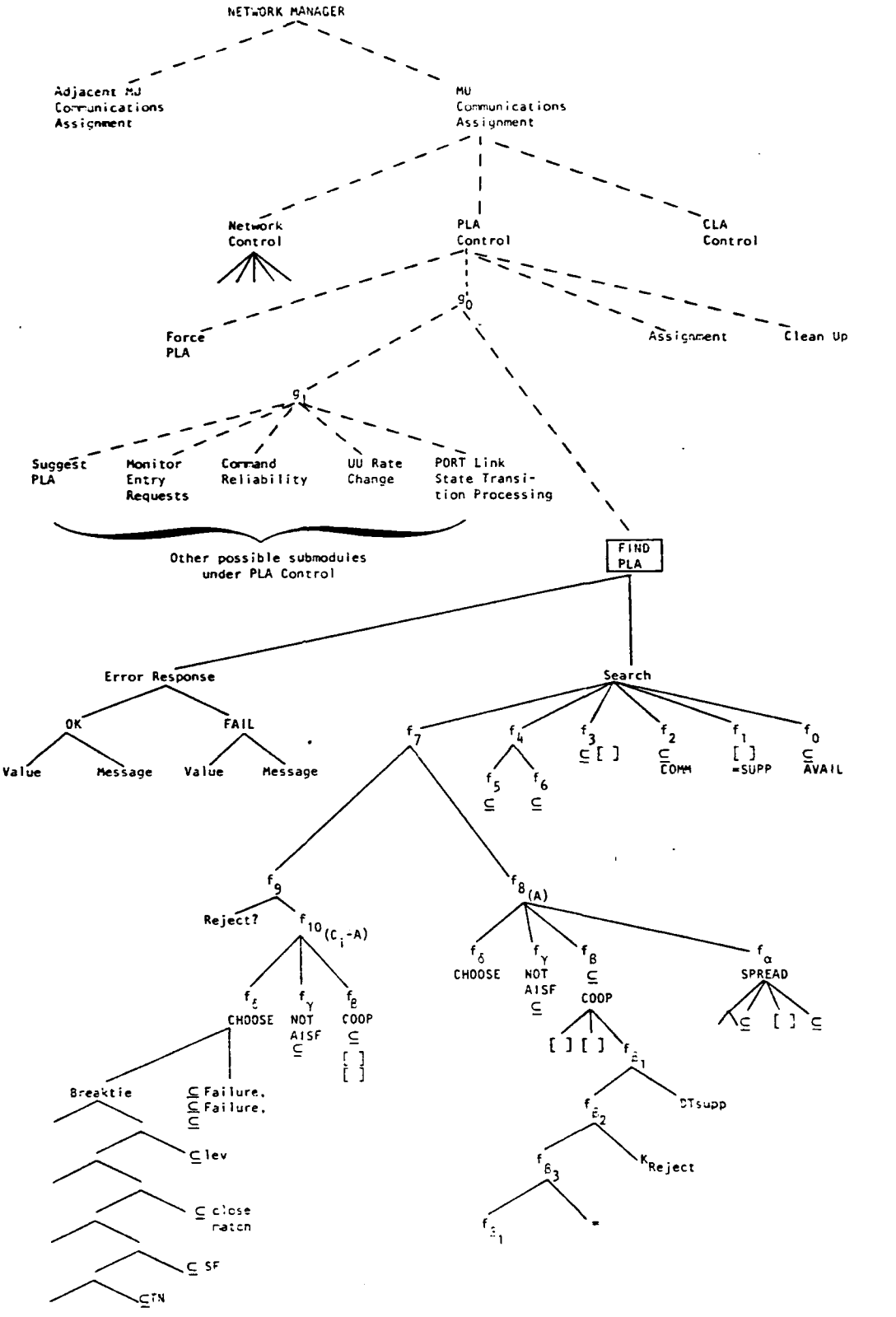
Those operations would then in turn be used to specify the FORCE-PLA module. Similarly, other modules such as Suggest PLA and Check Command Reliability, could be specified, adding operations and axioms in this fashion.

4.0 THE FIND-PLA MODULE

4.1 Identification of Submodules

Let us now turn to the control map of the Network Manager. Although we made a rough preliminary control map (following the Hughes documentation) in the preliminary stages of our investigation, this only covered in a general way the first three or so levels of detail (compare Figure 2.2-4 and Figure 4.1-1). In order to show clearly how a fully worked-out HOS specification including control map and accompanying AXES statements would look, we decided to restrict our attention for the purposes of this example to the most complex and interesting submodule, PLA Control. PLA Control module is not really decomposed in the Hughes PPS in great detail with respect to its submodules, but there are descriptions of some of its main functions. We can safely assume that there are at least three major submodules: (1) the one which designates a particular PORT Link Assignment for replacement; (2) the search algorithm itself, which finds a new Logical Time for possible PORT Link Assignment to a given UU; and (3) a housekeeping, or clean-up, module which takes care of rearranging the rest of the network once the new PORT Link Assignment has been made (e.g., changing the assignments of UUs supported by the UU whose PLA was changed, sending the change to the CLA-Control module so that CROSS Link Assignments can also be corrected, and so on).

Under the suggested Hughes module PLA Control, we further restricted our attention to one submodule, so as to demonstrate what a completed specification would look like. (A module is completely specified when all of the functions have been related by control structures and defined down to the point of primitive operations on defined data types. Of course non-primitive operations and control structures may be used, if they have been specified separately.) The module chosen was the search algorithm in submodule (2), FIND-PLA. It is related roughly, as shown in the hypothetical control map (Figure 4.1-1). The dotted lines indicate our best hypothesis as to how the system would relate these various modules. Since the operator system is discussed only sketchily in the PPS, there was not really enough information to make more than an educated guess



16 LEVELS WITHOUT ANY SET-TEST OR SET-FUNCTION RECURSIONS

Figure 4.1-1

at this. To specify this completely would require more extensive discussion with the designers at Hughes. Hence we have shown in this part of the control map only some main submodules without indicating inputs or outputs (also partially because of the difficulty in determining these from the information available, as discussed in Section 2).

In view of the complexity of the specification for the FIND-PLA mdoule and the similarity of the AXES statements accompanying the control map to coding in various higher-order languages, it should be reiterated here and emphasized that this is, of course, NOT code, but specification; it is implementation independent, even to the point of not specifying exactly HOW the sets of Logical Time are to be stored or referenced (e.g., lists, arrays, etc.) but rather as any kind of ordered set, leaving it up to the programmer to decide how the searches might be most efficiently conducted; this specification simply states which tests have to be conducted.

It should also be remarked that, as a pilot project and kind of pedalogical example of HOS specification techniques, this particular version of the control map for FIND-PLA is not to be construed in any way as final or "the last word," but rather as a tool for making the assumptions about the program specification very explicit so that they can more easily be examined and revised.

4.2 FIND-PLA Control Map

4.2.1 The Top Level of FIND-PLA

The section of the PPS which roughly corresponds to the FIND-PLA module is in Section 3.4.2.2.2.3 on page 3-34, "PORT link assignment selection" [11] (Figure 4.2.1-1). Much of the information needed to complete the specification of this submodule, of course, is not found in this section, but is either elsewhere in the PPS, in the STD, implicit in some of the discussion, or even absent entirely. In the latter case, we have tried to make reasonable hypotheses about what the system would do in order to complete the specification. Also there is a fair amount of redundant or unnecessary information in Section 3.4.2.2.3 [11] which

3.4.2.2.2.3 PORT link assignment selection. PORT link assignment selection shall, unless requested otherwise, determine the best match between UU desired PL states and available PLAs. Available PORT link assignments shall be determined from the unassigned PLAs directly supported by active UUs. When no PLA is available for assignment to a UU designated for PL state change, that UU shall be requested to demand entry. The PLA selection criterion and PLA restrictions are defined in the following subparagraphs.

3.4.2.2.2.3.1 PLA selection criterion. The best match between the desired PL state and the available PLAs shall be selected using the following order:

a. PLA with an exact match between the desired and available rate

b. PLA where the desired rate is lower than the available rate

c. PLA where the available rate is lower than the desired rate

For otherwise identical PLAs, selection shall be made in the following preference order:

d. PLA with lower levels

e. PLA with a closer rate match to the desired rate

f. PLA with earlier start frames.

3.4.2.2.2.3.2 PORT link restrictions. Available PLAs shall not be considered for assignment if the PLA either is supported by a UU that already cooperates in a PLA with the specified UU, or is active in frames that coincide with the specified UU's other assigned PLA. Whenever the MU is the supporting unit of an available PLA, all unassigned A-level PLAs shall be considered.

Each PLA shall be supported by only one UU. No two UUs shall have the same PLA. A PLA previously assigned to a UU shall not be available for reassignment until its deassignment or replacement has been explicitly acknowledged by:

a. A PLA command acknowledge

b. A mode command acknowledge

c. UU time out.

The A-level PLAs shall be assigned to different trees until all allocated trees have been used at least once.

does not pertain to the search algorithm. All of this will be discussed after a walk through the control map for FIND-PLA with commentary on its various sections.

Note again Figure 4.1-1, which shows roughly, without indicating inputs and outputs, how FIND-PLA is related to some of the other Network Manager functions. Various functions such as UU rate, Suggest PLA, or Monitor can request a new Logical Time for PORT Link Assignment by outputting T_{old} to FIND-PLA; FIND-PLA then outputs a new Logical Time, t_{new} to the PORT Link Assignment module and clean-up module, which changes the PLAs of the UUs supported by u_i via t_{old}.

Let us focus our attention on the top level of FIND-PLA, shown in Figure 4.2.1-2. The inputs to FIND-PLA are:

- p (the recommended Period (i.e., 1/cycle rate) (c.f. Footnote No. 2.
- t_{old} (the Logical Time of the PLA designated for replacement)
- T (the set of possible Logical Times)
- C_i (the set of communicants of the U_i which is assigned to t_{old})

The top level consists of just the main SEARCH submodule and the ERROR RECOVERY, which simply says to output a new Logical Time, t'_{NEW}, if one is found, and output the old one, t_{OLD}, if SEARCH does not find a new Logical Time for PORT Link Assignment, i.e., if t'_{NEW} = REJECT. In each case, provision is made for an appropriate I/O message to be output[8]. Note that the FIND-PLA module does not make PORT Link Assignments! What it does is to find a Logical Time which is appropriate for assignment to a given UU, which would either replace an old assignment, or in case of t_{OLD} = REJECT (i.e., the UU has no PLAs), could be an initial assignment. In other words, this module simply determines that a particular Logical Time meets certain specified requirements and is therefore appropriate to be used for a new PORT Link Assignment. The actual assignment would be done by the CLEAN-UP or housekeeping module of PLA-Control; the new Logical Time, t_{NEW}, and the given UU, u_i, would be inputs to the CLEAN-UP module, which would not only replace t_{OLD} by t_{NEW}[9], but also replace the Logical Times of the PLAs of any other UUs supported by u_i.

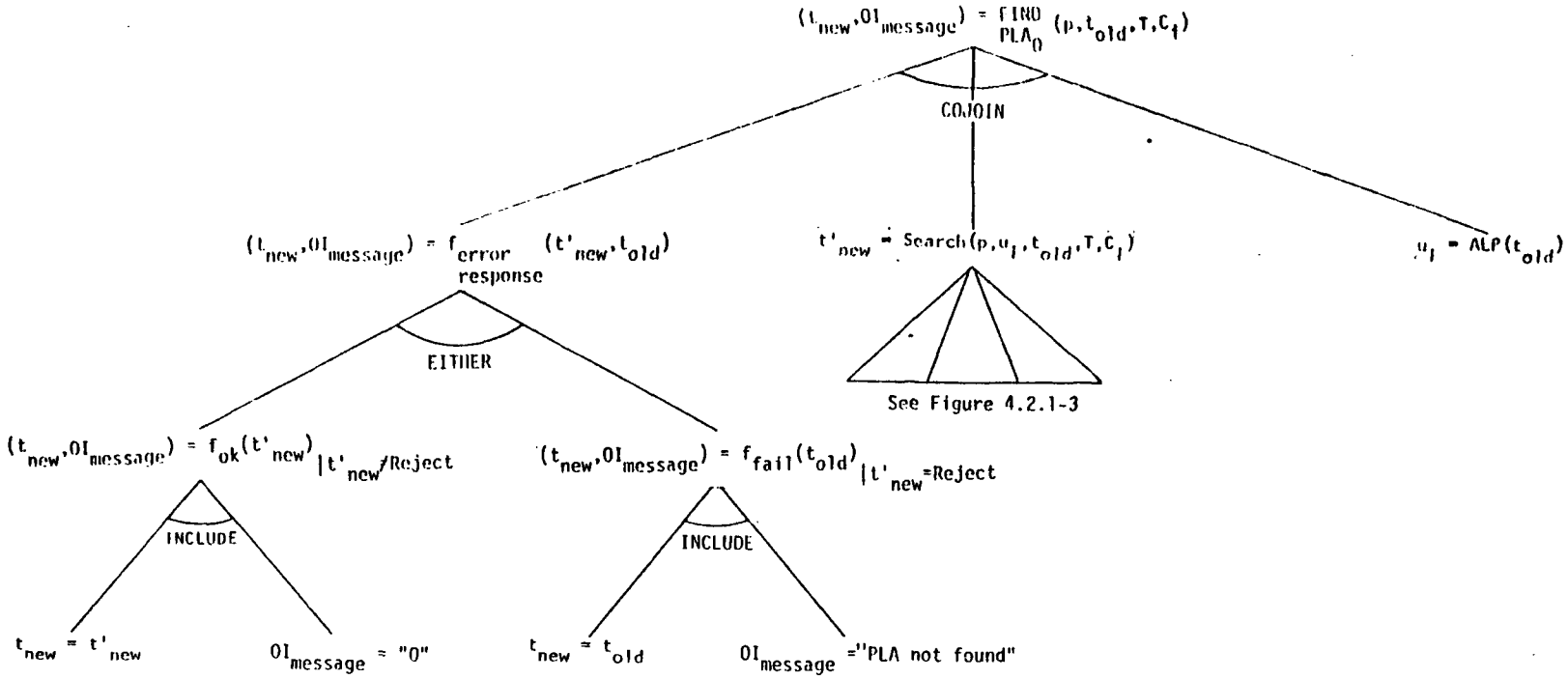

$(t_{new}, OI_{message}) = FIND\ PLA_0\ (p, t_{old}, T, C_1)$
COJOIN
$(t_{new}, OI_{message}) = f_{error\ response}\ (t'_{new}, t_{old})$
$t'_{new} = Search(p, u_1, t_{old}, T, C_1)$
$u_1 = ALP(t_{old})$
EITHER
See Figure 4.2.1-3
$(t_{new}, OI_{message}) = f_{ok}(t'_{new})\ |t'_{new} \neq Reject$
$(t_{new}, OI_{message}) = f_{fail}(t_{old})\ |t'_{new} = Reject$
INCLUDE
INCLUDE
$t_{new} = t'_{new}$
$OI_{message} = "0"$
$t_{new} = t_{old}$
$OI_{message} = "PLA\ not\ found"$

Figure 4.2.1-2

Now the search algorithm itself is shown in Figure 4.2.1-3. Before going through the various stages in detail, it would help to have an intuitive understanding of how it is done, as well as some comments about assumptions we have made. First of all, the way we assume the search is conducted is illustrated in Figure 4.2.1-4. The basic idea is to start out with a (perhaps large) set of Logical Times which could be possible candidates for PORT Link Assignment to the UU, u_i; by means of the various tests and requirements imposed by Section 3.4.2.2.3 of the PPS and elsewhere in the Hughes documentation to whittle down the set of possible candidates, throwing out the ones which are illegal for one reason or another until there remains a set containing only one unassigned Logical Time, t'_{NEW}, which will be the best match possible. That is, in terms of Figure 4.2.1-4, one starts out with a large number of candidate Logical Times for PORT Link Assignment, the set T, and after various tests and eliminations (the double arrows), one ends up with a single t_{NEW}.

It turns out that there is one "hitch" in this otherwise very straightforward process. For each particular UU, u_i, we only allow those Logical Times which are <u>T-supported</u> by Logical Times which happen to be PORT-Link-Assigned to other communicants of u_i (Figure 4.2.1-5). Note that the result of the new PORT Link Assignment is to have u_i <u>U-supported</u> by a different UU than the one which <u>U-supports</u> it currently. Thus, although the bulk of our tests on possible Logical Times are stated in terms of conditions on Logical Times, we have to refer, at least initially, to the UU's which are communicants of u_i (i.e., those which u_i has heard from at some time, indicating they are within radio range). This is shown schematically in Figure 4.2.1-5. Suppose all the boxes in the top row ($T_{Unassigned}$) represent unassigned Logical Times (which are therefore possible candidates for PORT Link Assignment to u_i). We can only choose the ones which are T-supported by Logical Times which are PORT-Link-Assigned to one or another of u_i's various communicants (UU's in the set C_i). But this is not a case of simple matching up, since each communicant UU in C_i has <u>two</u> PLA's because of duplexing. Therefore, one has two alternatives:

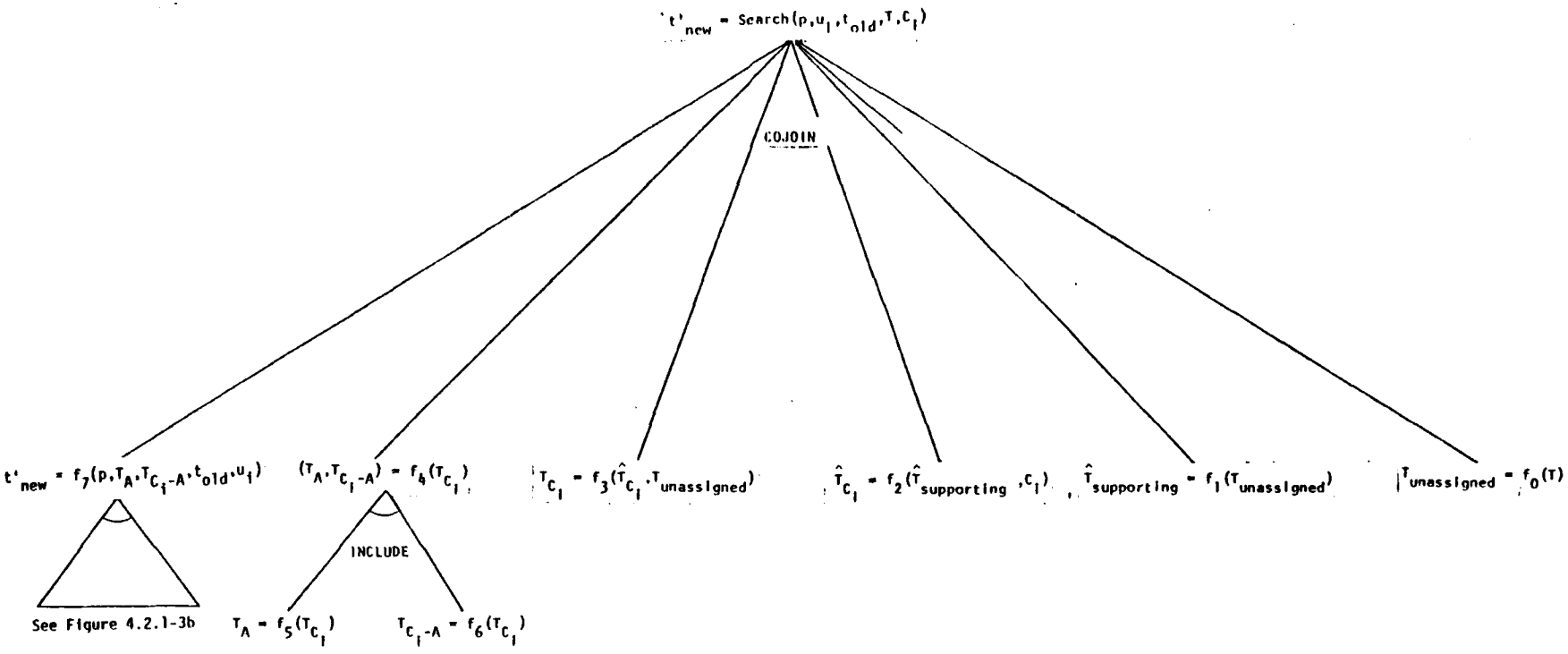


Figure 4.2.1-3a

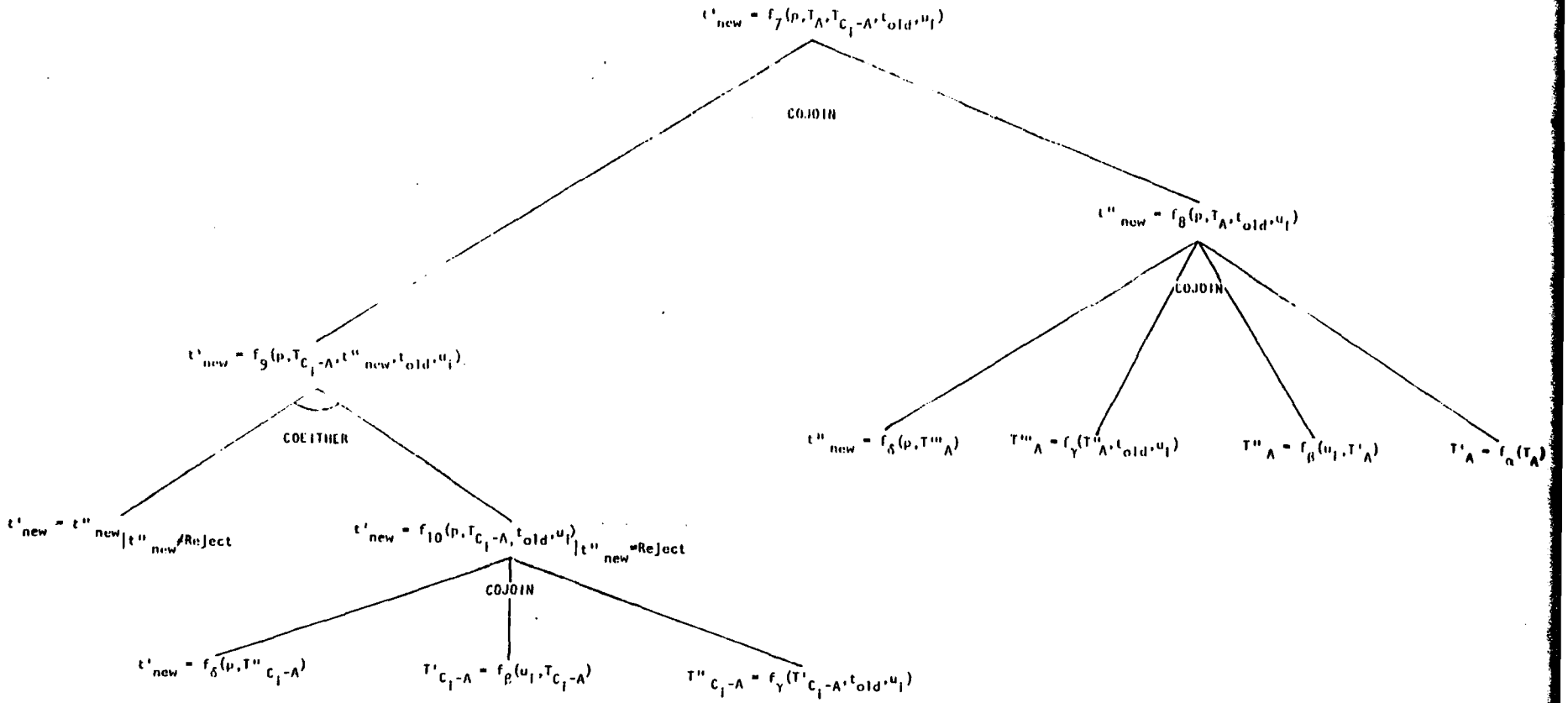


Figure 4.2.1-3b

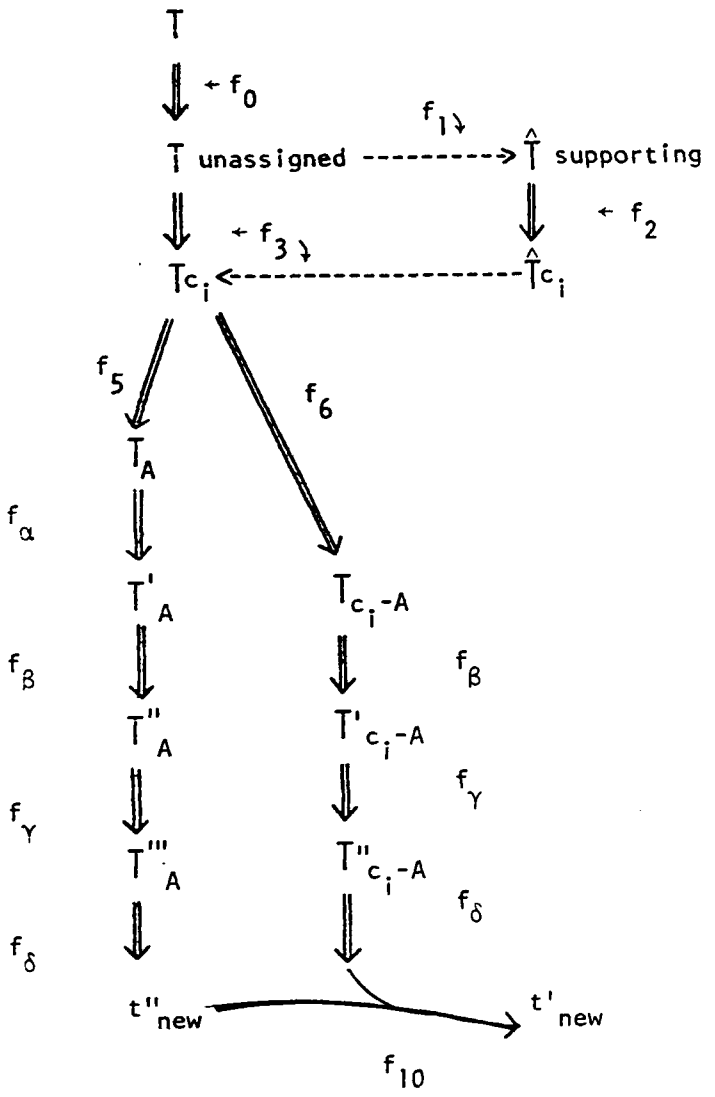

T
f_0
f_1
T unassigned
$\hat{T}$ supporting
f_2
f_3
T_{c_i}
$\hat{T}_{c_i}$
f_5
f_6
T_A
f_α
T'_A
T_{c_i}-A
f_β
f_β
T''_A
T'_{c_i}-A
f_γ
f_γ
T'''_A
T''_{c_i}-A
f_δ
f_δ
t''_{new}
t'_{new}
f_{10}

Figure 4.2.1-4

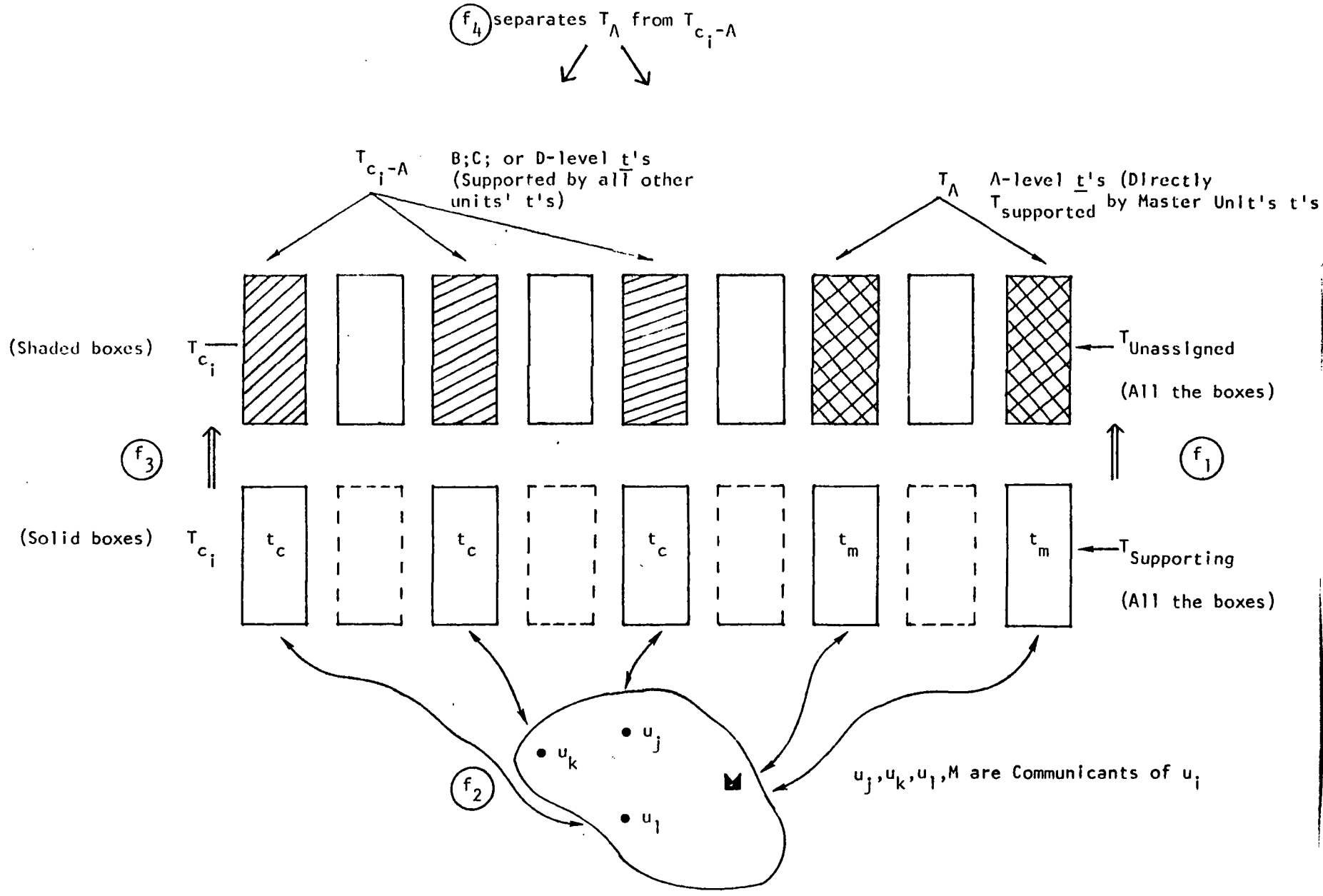

f4 separates TΛ from Tci-Λ
Tci-A
B;C; or D-level t's (Supported by all other units' t's)
TΛ
Λ-level t's (Directly Tsupported by Master Unit's t's
(Shaded boxes) Tci
TUnassigned
(All the boxes)
f3
f1
(Solid boxes) Tci
tc
tc
tc
tm
tm
TSupporting
(All the boxes)
uj
uk
ul
f2
uj,uk,ul,M are Communicants of ui

Figure 4.2.1-5

(1) to find all the Logical Times associated with the UU's in C_i, find which further set of Logical Times these Logical Times T-support, then find which of those are unassigned, i.e., available for assignment.

(2) to find the unassigned Logical Times, see which Logical Times T-support them, then eliminate all those which are not PORT-Link-Assignments of the u_i's communicants, C_i.

Either way is round-about, but the second seems to involve less computation, so we have opted for (2). Thus, in Figure 4.2.1-5, we first find $T_{Unassigned}$, the top row of boxes; then we find $\hat{T}_{Supporting}$ (all the boxes in the bottom row, which represent the Logical Times that support $T_{Unassigned}$. We then eliminate all the Logical Times not assigned to some communicant of u_i, (the dotted boxes) being left with the boxes labeled t_C or t_M. (We also need to treat possible A-level assignments differently than other levels. This has been shown by separating the $\hat{T}$ into those which are assigned to the Master Unit and those which are assigned to other user units; we wait until later in f_4 to actually separate out the T's which have Lev(t) = 0).

We then need to find the set of Logical Times which are both unassigned, and supported by the $\hat{T}_{C_i}$. This gives us the Logical Times indicated by the shaded boxes in the top row. We can now perform all the tests and matching required to choose one which will be the t_{NEW}.

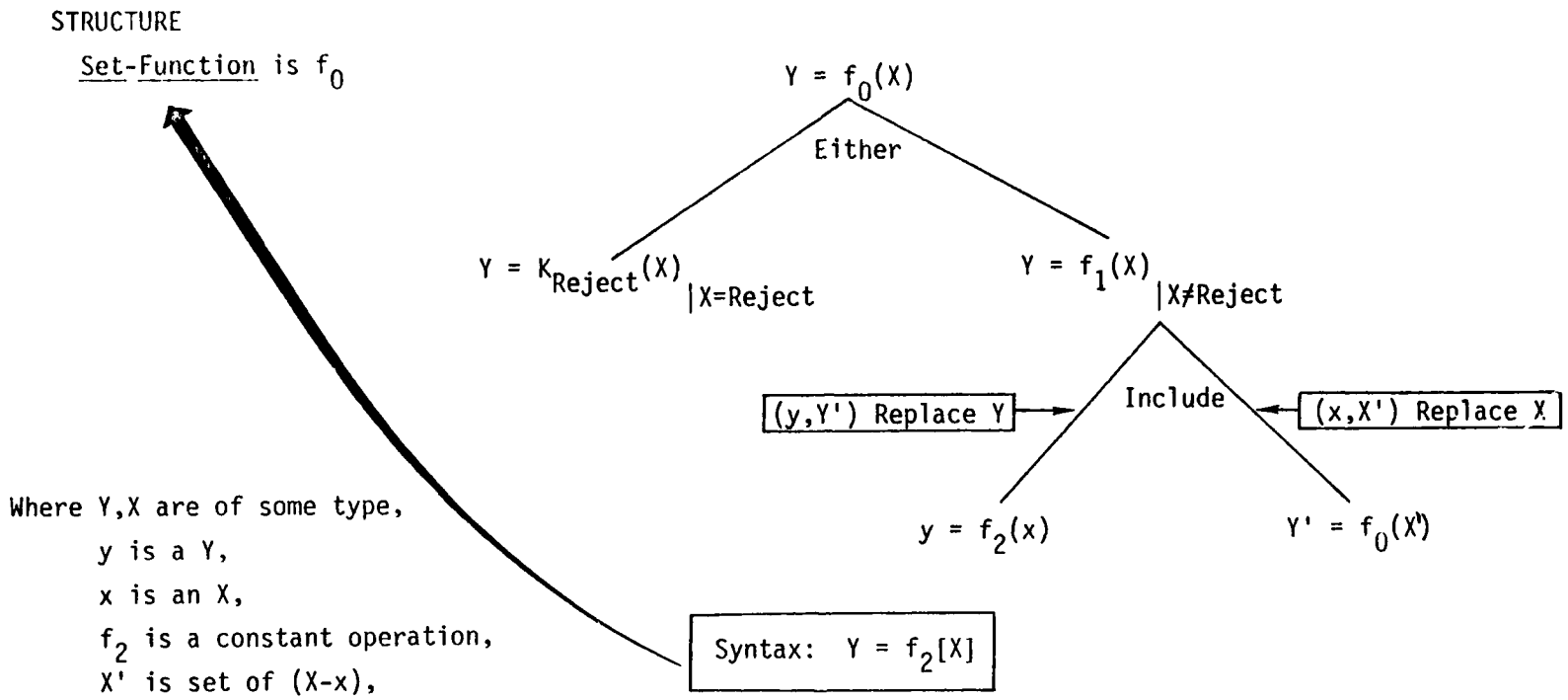
4.2.2 The Operations SET-FUNCTION and SET-TEST

Before discussing the exact statement of the functions in the control map for FIND-PLA, we need two additional tools, which HOS supplies us with. In large systems (programs) there are, of course, functions which are repeated over and over, and we may wish to simply define them in one place (especially if the definition is complex) to be invoked whenever needed. Similarly, HOS control maps and accompanying AXES statements can be used (aswe have seen in the section on data types) at the specification layer to define operations which can be referenced whenever needed.

In addition, just as there may be repeated functions, there may also be certain control structures which are used over and over again, but with different functions. AXES also gives us the capability to define these as needed and create a syntax so they can be invoked, employing the function needed for a particular case, in various places in the system. Examples of such structures are *Cojoin* and *Coinclude*, which are already HOS library structures (c.f. Appendix II).

For this PLRS module, we have defined two structures which will see use repeatedly. (They are, incidentally, used in other PLRS modules, and of wide general application. Because of their general applicability, they will be added to our library of AXES structures.) The first is called SET-FUNCTION, and is defined in Figure 4.2.2-1. What it does is to take as input any set-like data type (lists, arrays, files, etc.) and apply some operation which is individually valid on the members of the input set, to produce an output set of the same dimensions as the input set, but whose members are of whatever data type is normally output by the operation chosen. For example, suppose we take the operation y = sin(x), which takes as its input an angle and outputs the sine of that angle. But suppose that our input data will consist of 100 angles, stored in a 10x10 array. Then we simply write Y = sin[X] with square brackets and capital letters. If X is the 10x10 array, then SET-FUNCTION is invoked, which applies sin(x) to each angle, $x_{i,j}$, in the array X, producing a 10x10 array, Y, each of whose members, $y_{i,j}$, is the sine of the corresponding $x_{i,j}$. Since we are continually working with large numbers of Logical Times, presumably sorted in some kind of ordered set (list, array, etc.), this will clearly be a useful control structure for the FIND-PLA module.

The other necessary control structure is SET-TEST, defined in Fig. 4.2.2-2. This is different from SET-FUNCTION, which really transforms one set into another, in that the output is always a subset of the input. What it does is to take some set of objects, and throw out all those which don't meet some test or criterion. For example, if we had a large set of Logical Times, we might want to consider only those whose Period was, say, less than 8. Now the syntax for SET-TEST is: "T_2 *is formed from* T_1 *forall* g(p.t)", where T_1 is the set we start out with, and T_2 is the set we have after eliminating all the t's in T_1 for which the boolean function



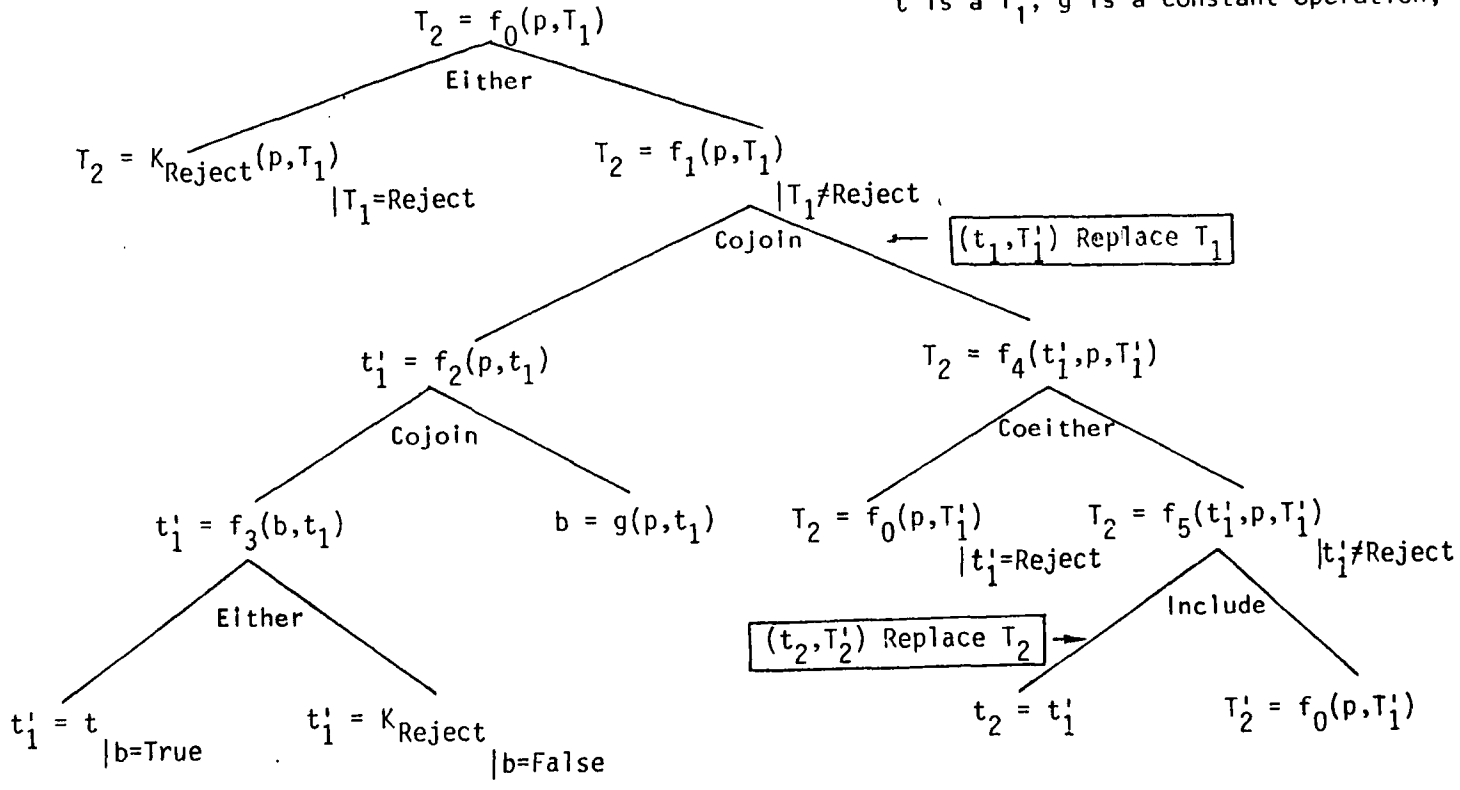
WHAT IT DOES: for any operation valid on a set, it applies the operation to each element and forms a new set of the same size, e.g., if B and A are both mxn arrays, and $a_{i,j}$ is some angle, then B=sin[A] means each member of $b_{i,j}$ of B is $b_{i,j}=\sin(a_{i,j})$.

Figure 4.2.2-1

where T_2, T_1 are of some type,
t is a T_1,
p is of some type,
b is a boolean,
T_1' is set of (T_1-t),
t is a T_1, g is a constant operation;

STRUCTURE

Set-Test is f_0



Syntax: T_2 is formed from T_1 for all $g(p,t)$.

Figure 4.2.2-2

g(p,t) = FALSE. So for our example, we could say, where T_2 and T_1 are some kind of set of Logical Times, "T_2 *is formed from* T_1 *forall* Per(t) < 8." This says that the times, t, which are in T_2, are those which were in T_1, and whose period was less than 8. In the figures in this report, we sometimes replace the words "is formed by" by the symbol "$\subseteq$" for brevity. Notice that the statement "Per(t) < 8" corresponds to "g(p,t)" in the general syntax for SET-TEST. The operation g is really any (n+1) place boolean operation (i.e., whose value is either TRUE or FALSE) and whose first n places are fixed-parameters or constants, and whose (n+1)th place is the variable being tested. That is, g(p,t) is just the statement of some possible property of t, where p is a list of some constant parameters which may be necessary to state the property, e.g., "8" in the above example.

To reiterate, in the statement "Per(t) < 8," the constant 8 corresponds to the p in "g(p,t)," and the compounded operation "Per(_) < _" corresponds to the "g(_,_)". This is perhaps easier to see in prefix-notation than in infix-notation: "< (Per(t),8)."[10]

At any rate, it is clear that if we are to begin with a set (perhaps large) of logical times which are possible candidates for PORT Link Assignment and find a single "best" candidate by throwing out all those which don't meet certain criteria, then this SET-TEST structure will be the basis for this module. Once we have defined these two structures for working with sets, then it is simply a matter of trying to state the tests and criteria to be used correctly, and in the correct order.

4.2.3 The Search Algorithm

Now that we have these two structures which can be invoked at will to either perform an operation on a set or to perform a test on a set, let us look again at the search algorithm (Fig. 4.2.1-3). The function names f_0 -f_5 and f_α - f_δ are used to save space and are abbreviations for the full specifications written out in Figure 4.2.3-1.

The first function, f_0, takes the set of all possible Logical Times as input and yields the ones which are unassigned as output. It simply applies the function ALP (the inverse of PLA, PORT Link Assignment)

f_0: $T_{unassigned} \subseteq T$ forall Reject = ALP(t)

f_1: $\hat{T}_{supporting} = DTsupp[T_{unassigned}]$

f_2: $\hat{T}_{c_i} \subseteq \hat{T}_{supporting}$ forall Element?(APL(t),C_i)

f_3: $T_{c_i} \subseteq T_{unassigned}$ OR (DTsupp?($[\hat{T}_{c_i}]$,t))

f_4: $(T_A, T_{c_{i-A}}) = f_4(T_{c_i})$ SEE BELOW FOR DETAILS

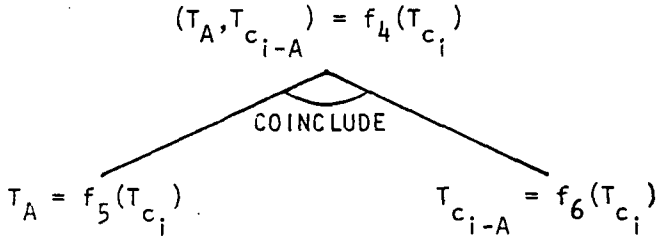
f_α: $T'_A = f_\alpha(T_A)$ SEE Figure 4.2.3.2-1b

f_β: $T'_{c_{i-A}} = f_\beta(T_{c_{i-A}})$ SEE Figure 4.2.3.3-1

f_γ: Operation: $T_2 \subseteq T_1$ forall NOT(AISF(Other(u_i,t_{old}),t))
where T_2,T_1 are sets of logical times; u_i is a uu; t_{old}, t, are logical times:

f_δ: Operation: CHOOSE(p,T) SEE Figure 4.2.3.4-1

f_4:



f_5: $T_A \subseteq T_{c_i}$ forall Lev(t) = 0

f_6: $T_{c_{i-A}} \subseteq T_{c_i}$ forall Lev(t) > 0

Figure 4.2.3-1

to each $\underline{t}$ in T, and if it is not assigned, i.e., if ALP(t) $\neq$ y, for some $\underline{u}$, then ALP(t) = REJECT. Thus, all this function has to do is find the t's such that ALP(t) = REJECT.

4.2.3.1 Finding Eligible Logical Times

Remember that this is the specification layer, not the implementation layer; f_0 does not necessarily say how, exactly, PORT Link Assignments are to be stored or referenced, or even that $T_{Unassigned}$ must be calculated by going through the list of all logical times on each pass. At the implementation layer, for example, $T_{Unassigned}$ might be stored in some temporary memory and simply updated on each pass. We might even want to revise the specification and make f_0 part of the CLEAN-UP module, i.e., input $T_{Unassigned}$ to FIND-PLA directly.

Note how straightforward such a change would be. We can see, in the control map, exactly which input variables would have to be changed, and we would get a new output variable, $T_{Unassigned}$ from CLEAN-UP. This illustrates two points: (1) a property of HOS specifications: that changes in the specification are relatively easy to make prior to implementation because of the modularity of the control map format; and (2) a property of the PLRS system: it is cyclic, rather than strictly linear; i.e., the output of a function "late" in the cycle may produce an output which serves as one of the inputs on the next cycle. For example, an ordering of functions like "f_0, f_1, f_2, f_3" might have the same effect as the order "f_1, f_2, f_3, f_0" since the output of f_0 will still serve as the input to f_1 on the next cyclic pass through the system. This is a matter deserving more attention than we have space for here, but since ordering is treated only implicitly in the PPS and STD, we simply note it as an issue to be resolved.

The next function, f_1: $\hat{T}_{supporting}$ = DTsupp $[T_{Unassigned}]$ simply gives us (c.f. Fig. 4.2.1-5) from the unassigned logical times, the set of logical times which support them (bottom row of time-slot boxes in Fig. 4.2.1-5. We then ask, via f_2, which of those logical times are PORT-Link-Assigned to UU's which are communicants of the u_i whose t_{OLD} we are trying to

change. That is, if t_s is a Logical Time in the bottom row ($T_{Supporting}$), is there a $u_s = ALP(t_s)$, such that u_s is also a member of the set of communicants, C_i? That is what f_2 states: $\hat{T}_{C_i}$ *is formed from* $\hat{T}_{Supporting}$ *forall* Element?(ALP(t),C_i)." Note that the condition on formation of the new subset is a compound function; this could have been written out in full AXES format as follows:

Operation: b = g(C_i,t);
  Where b *is a* Boolean,
  C_i *is a* Set (*of* UUs),
  t *is a* Logical Time,
  u *is a* UU;
b = Element?(u,C_i) *Cojoin* u = ALP(t);
END g;

The format "Element?(ALP(t),C_i) is simply an abbreviation for an AXES defined operation.

The function f_3 then asks of the Logical Times in the top row ($T_{Unassigned}$) are they T-supported by a Logical Time which is assigned to one of u_i's communicant's? (c.f. Figure 4.2.1-5). That is, it eliminates from $T_{Unassigned}$ all the Logical Times not supported by a Logical Time which is PORT Link Assigned to some communicant UU (unshaded boxes).

4.2.3.2 The A-Level Logical Times

The function f_4 simply separates these Logical Times in T_{C_i} (which are candidates for PORT Link Assignment) into the A-level ones (T_A) and all the rest (T_{C_i-A}), since the A-level candidates will be treated differently (f_α) and considered first (f_8) for PORT Link Assignment.

This separation of A-level logical times corresponds to the PPS statement: "Whenever the MU is the supporting unit of an available PLA, all unassigned A-level PLAs shall be considered." This statement is a bit problematical, for if the "MU is the supporting unit of an available PLA," then the implication is that the MU is one of u_i's communicants. Hence it follows

logically from the search algorithm as stated for the general case that any unassigned A-level Logical Time will be automatically among those considered for assignment. Thus, as worded in the PPS, the requirement is unnecessary; it could be taken to mean, "...all unassigned A-level PLAs shall be considered first." This seems like a not unreasonable requirement, since the A-level Logical Times are also singled out for a special test, and from the STD, the philosophy seems to be that one tries to fill in A-level "branches" in the trees first.

Our search algorithm itself, f_7, is arranged to take this into account, by first looking at the A-level Logical Times separately in f_8. Then if a $t"_{new}$ is found among the A-level Logical Times, it is made the new PLA, t'_{new}, by f_9. If no $t"_{new}$ is found among the A-level Logical Times, f_9 directs f_{10} to look among all the others (T_{C_i}-A) for a possible PLA.

The second way in which A-level Logical Times are treated differently is in being subjected to the test in f_α, which corresponds to the PPS statement, "The A-level PLAs shall be assigned to different trees until all allocated trees have been used at least once." That is, UUs shall be assigned to Logical Times with different tree numbers, until there is at least one PLA for every tree. (Incidently, note again that the sentence, as worded, is meaningless: if PLA is intended to mean "Logical Time," it is false, because Logical Times have a fixed tree number and cannot be "assigned" to a tree--they are already by definition in a tree. If PLA really does mean PORT Link Assignment, then a PORT Link Assignment can't be assigned; only UUs can be assigned to Logical Times and vice versa. Hence, it should read, "The A-level UUs shall be assigned..." or even more accurately, "If there are any trees all of whose Logical Times are unassigned, UUs should be assigned to A-level Logical Times in these trees first."

In less formal terms, in the initial stages of building the network, one wants to fill all the trees as quickly as possible, so that there are no "empty" trees, i.e., ones whose logical times are assigned to no UUs at all. Thus, if there are any "empty" trees still in the network, we must try to assign our UU_i to one of them first (and since they are "empty", it, of course, has to be an A-level assignment).

This is illustrated in Figure 4.2.3.2-1. The cross-hatched boxes represent Logical Times assigned to the MUwhich potentially T-support Logical Times at the A-level. In the picture, two of these (the black ones) have been assigned to UUs, one in Tree 1 and the other in Tree 5. Trees 12 and 17 have no UUs assigned to their Logical Time slots. Thus, if we have a UU requesting entry, we would want to assign it to, say, Tree 12.

This is what f_α does. Since it is essentially an existence test ("Do any empty trees exist?"), like all existence tests, it is somewhat cumbersome. $T_{initial}$ is the set of available Logical Times which happen to belong to empty trees. If there are any, function g_i sets $T'_A = T_{initial}$; otherwise, T'_A = the original T_A that we started with, and the computation continues in a normal fashion.

4.2.3.3 Restrictions on PLAs

Functions f_β and f_γ are requirements imposed upon the choice of Logical Times before we even begin the search. Function f_β insures that there are no loops--that the network of UUs and Logical Times is really tree-like." It is assumed that it is desirable not to have either "real" loops (in the same tree; see Fig. 4.2.3.3-1a) or "virtual" loops (the two UUs active in different trees, Fig. 4.2.3.3-1b). While this seems desirable from a practical point of view (we want a maximally spread-out netowrk, so that if one UU is knocked out of action, it will create a minimal disturbance of communication links), it does impose a severe restriction on the number of Logical Times available for PORT Link Assignment to a given UU. Also (like function f_α), it is rather complex computationally, as can be seen from the control map in Figure 4.2.3.3-1. While the Logical Times that don't meet the condition "not ...supported by a UU that already cooperates in a PLA with the specified UU" [ ], this condition is quite complex computationally, as we can see by its control map (Fig. 4.2.3.3-1a, f_β). (Again an existence condition!--"Does a UU exist such that one of its assigned Logical Times support the proposed Logical Time being considered for assignment to u_i and whose other assigned Logical Time is either T-supported by or T-supports the other Logical Time of u_i?")

INITIAL SPREAD ACROSS TREES

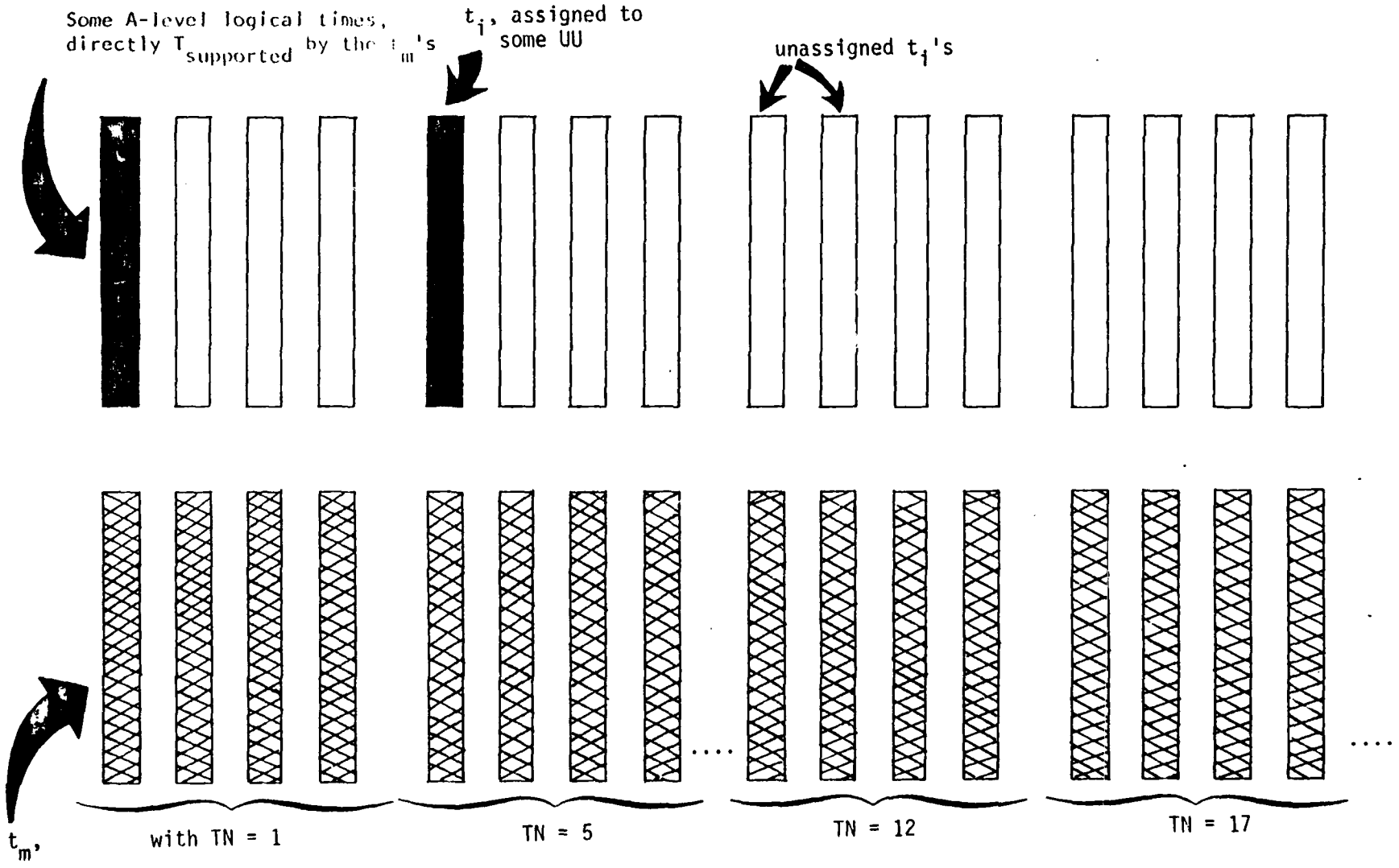


Figure 4.2.3.2-1a

INITIAL SPREAD ACROSS TREES

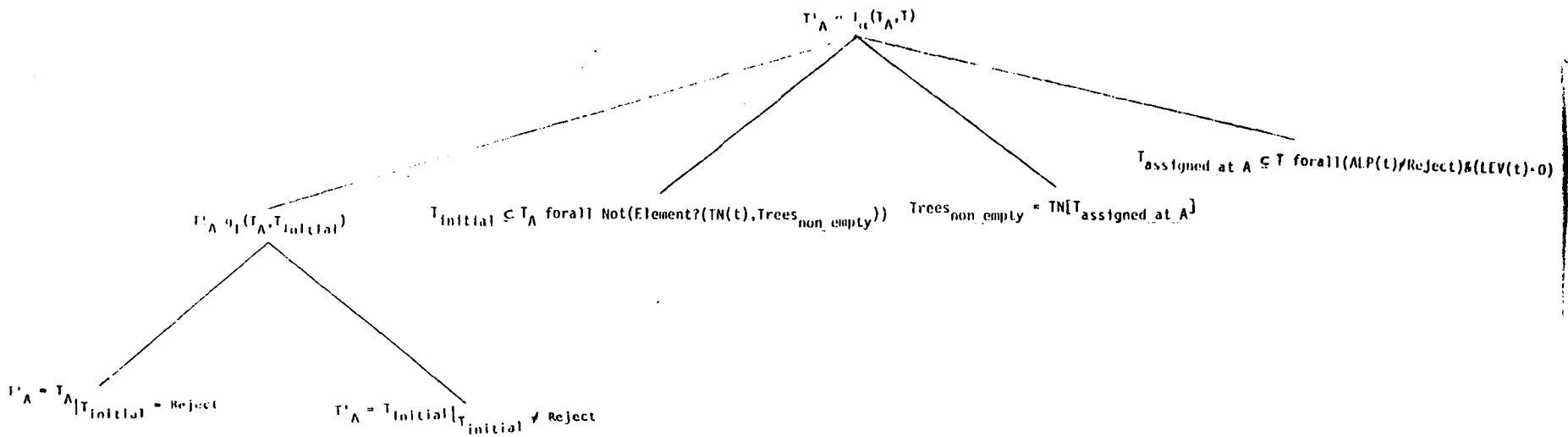
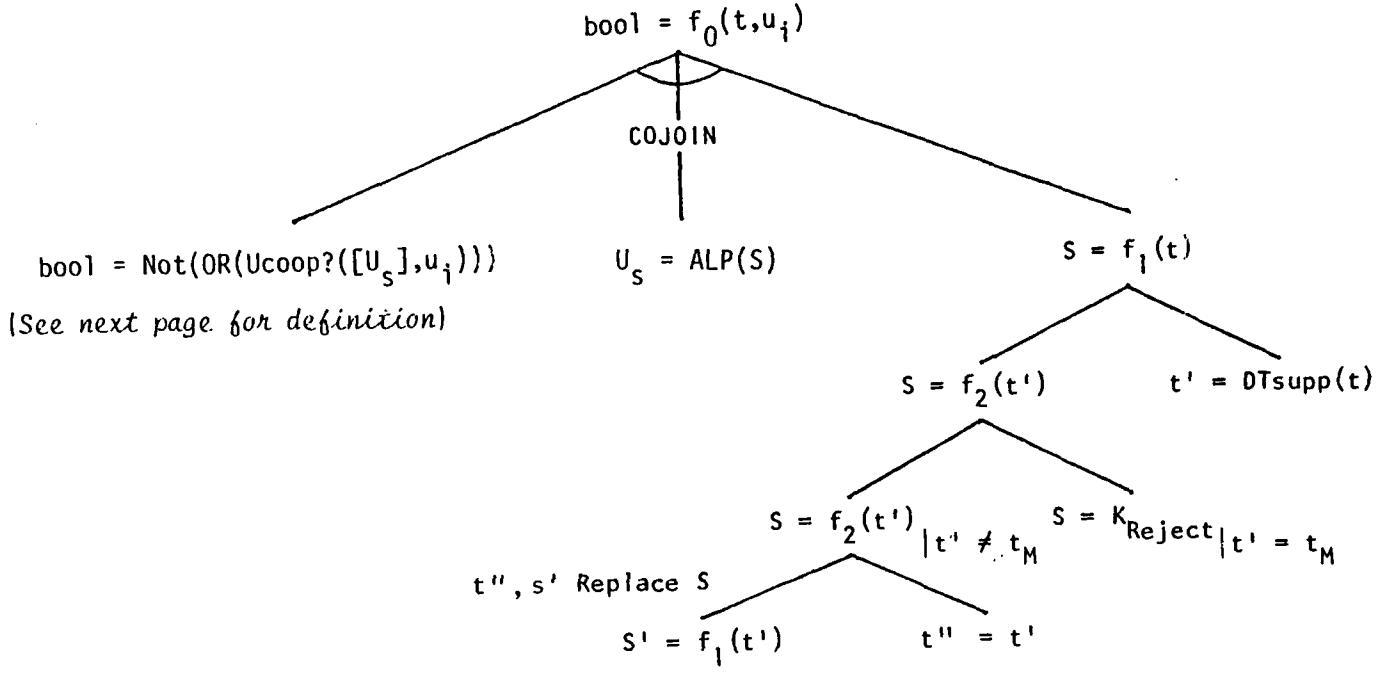


Figure 4.2.3.2-1b

f_β: $T'_{C_i\text{-}M} \subseteq T_{C_i\text{-}M}$ forall bool = $f_0(t,u_i)$



(See next page for definition)

"PLA not considered if ...supported by a uu that already cooperates in a PLA with the specified uu."

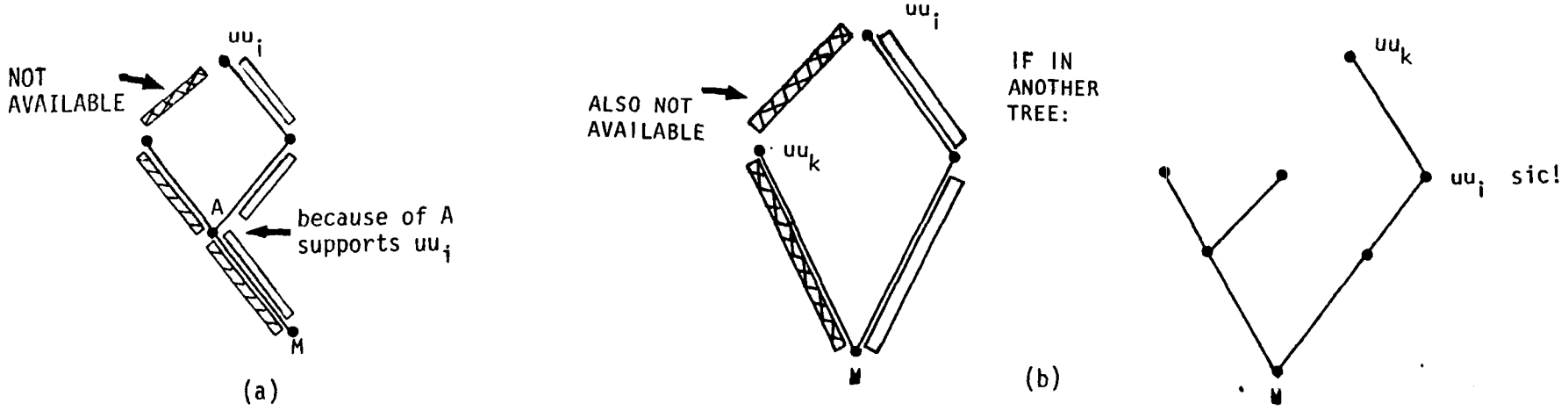


Figure 4.2.3.3-1

Definition of COOPERATE:

OPERATION: boolean = Ucoop?(u_1,u_2);

WHERE u_1,u_2 ARE UUs;

Ucoop?(u_1,u_2) = OR(Usupp?(u_1,u_2), Usupp?(u_2,u_1));

END Ucoop?;

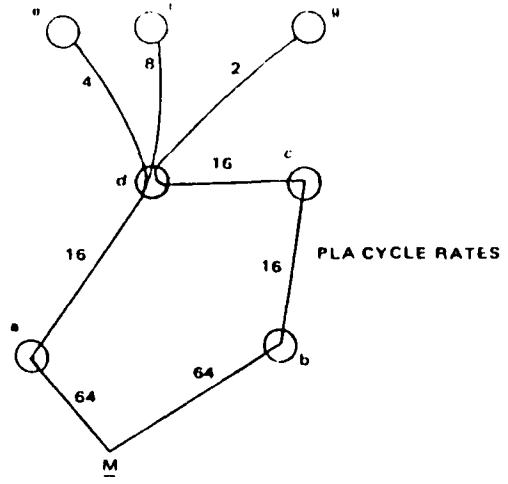
Figure 4.2.3.3-1
(con't)

Function f_γ does not seem as complex only because we have already defined AISF (Active-in-Same-Frame) as an operation in the section on data types. Furthermore, the requirement, as stated in the PPS, is that the Logical Time not be considered if it "is active in frames that coincide with the specified UU's other PLA" (emphasis added). Unfortunately, specifying other is not so easy as it might seem at first glance. because of the duplexing. To illustrate, consider an early problem that we ran into in understanding the PPS: Figure 4.2.3.3-2 (PPS Figure 3.4.2-2), we were surprised to find that although the PORT-Link supporting UU_g (going through UU_d) is the second PLA for UU_d (c.f. row d_2 in the chart, where the rate is entered in column APL2, not column ALP1), and yet the rate for UU_c was entered in column APL1, indicating that this Logical Time was the first PLA for unit UU_c. That is, there is no fixed concept of "other," i.e., of first or second PLA, as we have noted in the data-type specifications, if we say UU_1 supports UU_2 and $PLA(u_i) = t_a,t_b$, $PLA(u_2) = t_c,t_d$. We have no way of knowing in advance of actually checking all the possibilities whether t_a T-supports t_c, or if t_a T-supports t_c, or if t_b T-supports t_c, or if T-supports t_d! --four possibilities!

Hence, the address of the "other" PLA for some given UU is not a constant (like "FIRST" or "SECOND"), but must be referenced either by including the other PLA (e.g., the other Logical Time, t_{other}) as an input to the search algorithm, or by calculating it when we need it (Figure 4.2.3.3-3).

4.2.3.4 The Search Itself

Finally, we come to the search algorithm itself, which is fairly straightforward, compared to the other tests and conditions. This is done in Function f_δ. After seeing all the above complexity, it should be noted here, that when a simple condition is imposed (instead of the very complex conditions suggested by the PPS for the previous tests), the AXES structure SET-TEST allows a very simple and clear statement of the requirements (Figure 4.2.3.4-1). On the right-hand side of the Cojoin, we simply take the set we began with (T_1) and throw out all the Logical Times whose periods are not equal to the required period. If there are none left, T_2 = REJECT, and the FAILURE structure (an AXES library structure of Appendix II) allows us to try a different function, in this case looking for



	UNIT	START FRAME	PERIOD	RATE	LEVEL	TREE	SUPPORTS (AT ALL LEVELS)	DIRECTLY SUPPORTS	SUPPORTED BY (AT ALL LEVELS)	DIRECTLY SUPPORTED BY	APL1	TPL1	APL2	TPL2
	a	0	4	64	A	0	d_1,e,f	d_1	MU	MU	28	192	0	0
	b	2	4	64	A	0	c,d_2,g	c	MU	MU	34	192	0	0
	c	2	16	16	B	0	d_2,g	d_2	b,MU	b	18	32	0	0
•	d_1	0	16	16	B	0	e,f	e,f	a,MU	a	12	32	0	0
•	d_2	2	16	16	C	0	g	g	b,c,MU	c	0	0	2	16
	e	16	64	4	C	0	—	—	a,d,MU	d_1	0	4	0	0
	f	0	32	8	C	0	—	—	a,d,MU	d_1	0	8	0	0
	g	2	128	2	D	0	—	—	b,c,d_2,MU	d_2	0	0	0	0

• UNIT d HAS 2 PLAs

FRAMES

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	30	31	32	33	34	76	77	78	79	80	81	82	83	84	85	86
A	a		b		a		b		a		b		a		b		a		b		b		a		b	a		b		a		b		a		b
B	d_1		c														d_1		c				d_1		c					d_1		c				
C	f		d_2														e		d_2				f		d_2					e		d_2				
D			g																																	

I.E., UNIT IS ACTIVE IN A FRAME WHEN THE FRAME MODULO PERIOD = START FRAME

A Network Example

Figure 4.2.3.3-2

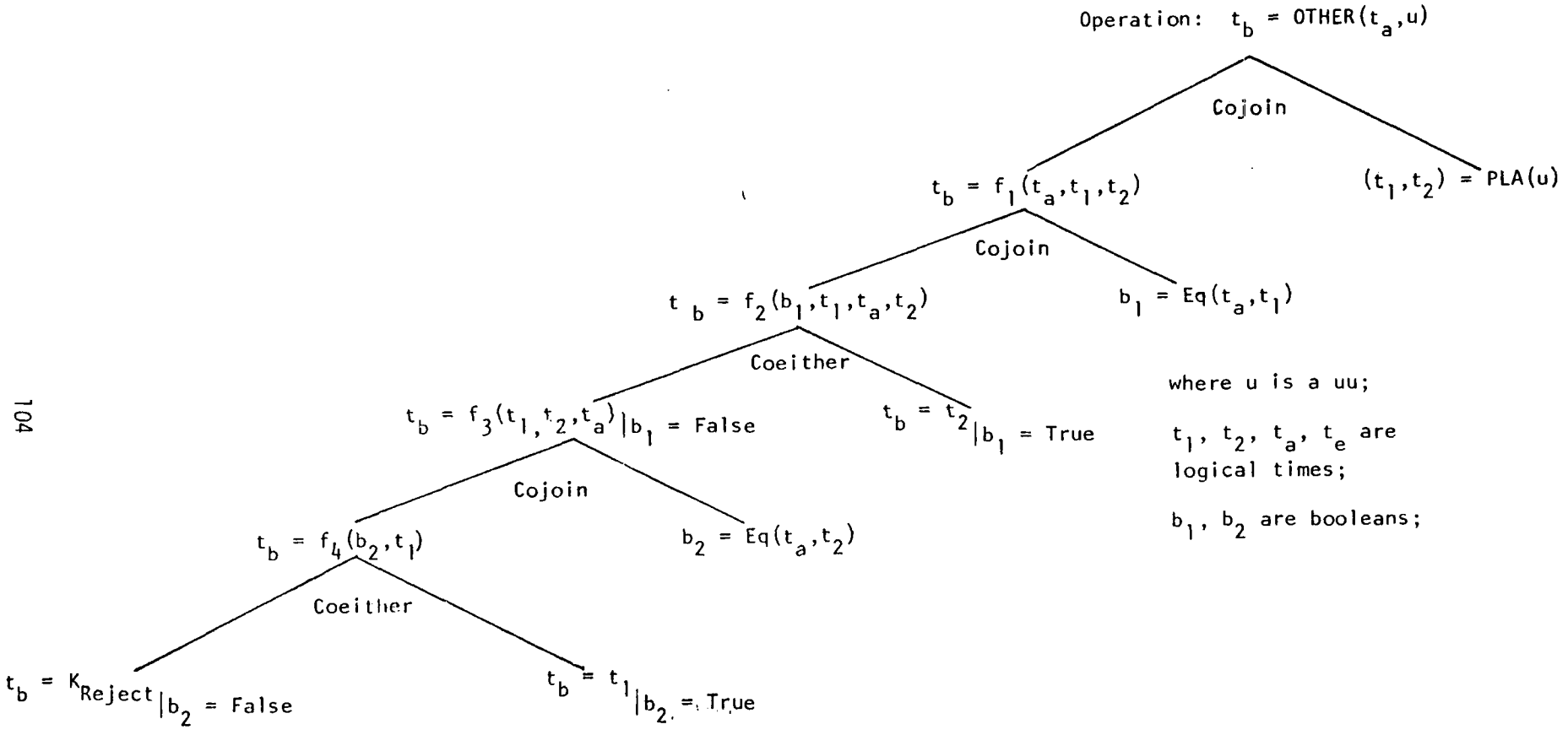


Figure 4.2.3.3-3

Operation: Choose

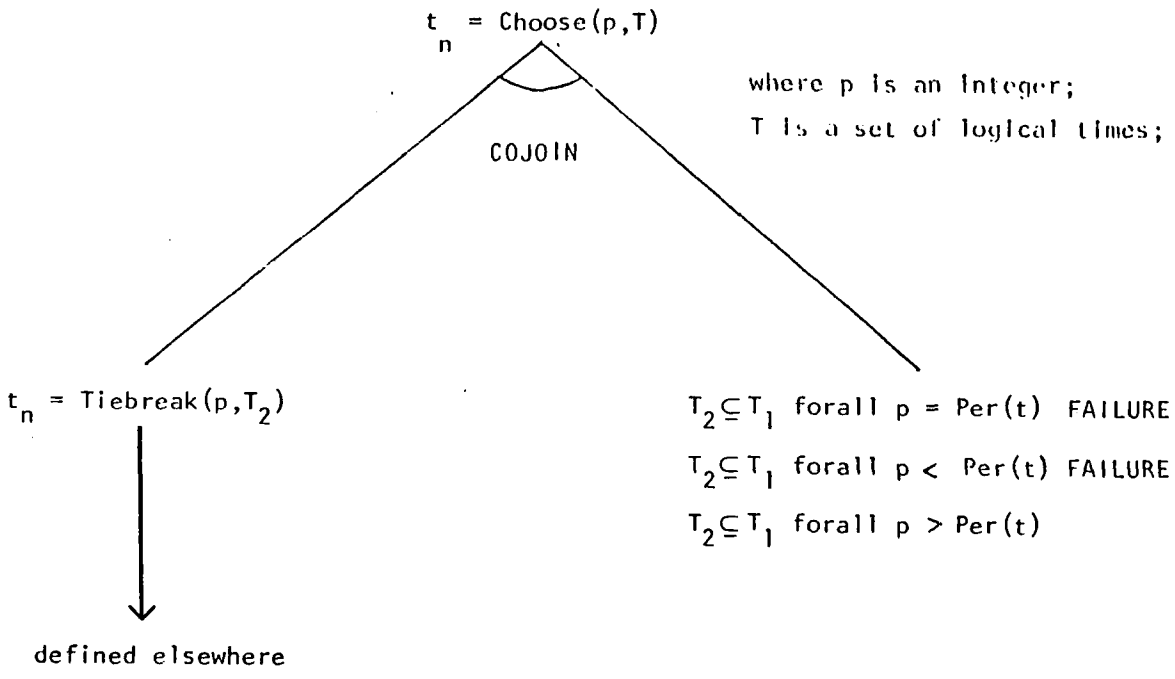


Figure 4.2.3.4-1

the Logical Times with periods less than the required period. If there are none, FAILURE again lets us try a third possibility--greater-than the required period. (Note that less and greater have been reversed because we are checking period, rather than cycle rate, c.f. comment in Footnote #2). We do not need a third FAILURE since the top level of Tie-Break will pass on a REJECT if there are no possible assignments.

The Tie-Break operation (Figure 4.2.3.4-2)is also fairly self-explanatory, except that, as given in the PPS, it does not break all possible ties. To see why this is so, remember that Logical Times are exhaustively defined by PERIOD, LEVEL, TREE-NUMBER, and START-FRAME. While given a particular start-frame, level, and tree-number, there can be only one period (because of Axiom 5 of data-type Logical Time), it is perfectly possible for there to be two Logical Times with the same period, level, and start-frame but different tree numbers. (The reader can demonstrate this by working through the Logical Times t_α - t_ε in Figure 4.2.3.4-3. There is no way to choose between t_α and t_γ.) We have suggested an arbitrary fourth test based on tree-number as part of the Tie-Break operation.

4.3 Conclusion

In summary, an exact specification of this module (FIND-PLA) in HOS terms would consist of the control map accompanying AXES statements (with perhaps a minimal explanatory commentary) and the data-type specifications, which would cover all the modules. This is what a rewrite of the PPS would consist of in HOS terms. A sample explanatory commentary for such a rewrite is given in Figure 4.3-1. It is interesting to compare this with the original PPS page 3-43 that we started out with. In order to turn this into a detailed and explicit specification (e.g., one that could be coded from in some relatively straightforward way), there were many problems to be resolved and hidden implications to be discovered. For example, Figure 4.3-2 shows some of the words used in a technical sense, just in this one section of the PPS (Sect. 3.4.2.2.3) along with some problems (discussed earlier) regarding their meaning. Similarly, Figure 4.3-3 highlights some of the functional problems: (1) there is no order among the operations and tests suggested (functions are marked

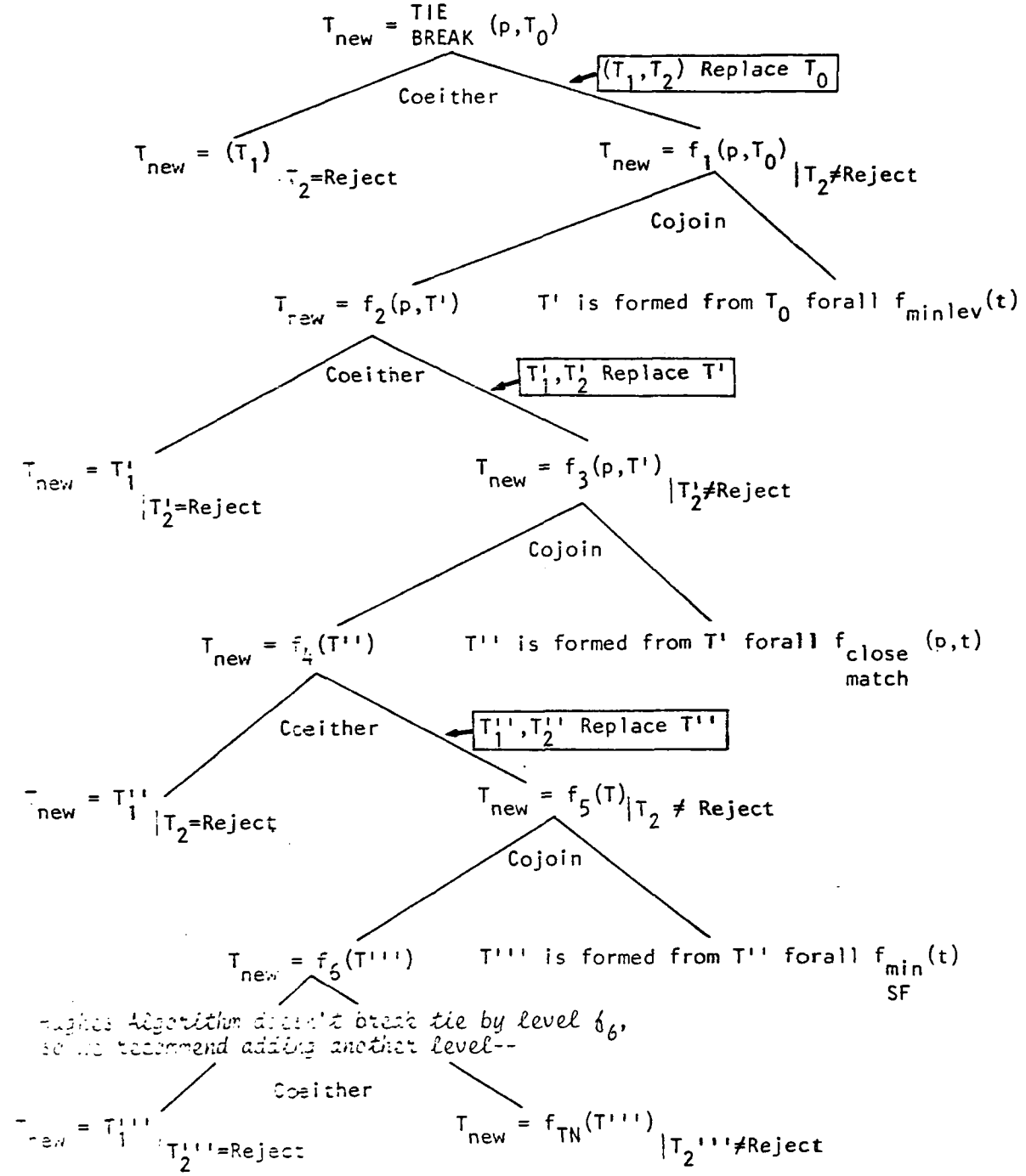


Figure 4.2.3.4-2

f_{minlev} = (Lev(t) = Min(Lev[To]))

$f_{close\ match}$ = (Abs(p-Per(t))) = Min(Abs(p-Per[T']))

$f_{min\ SF}$ = (SF(t) = Min(SF[T'']))

A possible f_{TN} = (TN(t) = Min(TN[T'']))

Why it doesn't break tie:

Suppose want per = 8, and

	per	Lev	TN	SF
t_α =	4 ,	1 ,	2 ,	2
t_β =	4 ,	1 ,	7 ,	3
t_γ =	4 ,	1 ,	13 ,	2
t_δ =	16 ,	1 ,	21 ,	7
t_ε =	4 ,	2 ,	7 ,	5
t_η =	32 ,	2 ,	2 ,	3

Figure 4.2.3.4-3

WORDS NEEDING DEFINITION:

PLA (PORT Link Assignment):

can mean either (1) the assignment process

e.g., "assignment of a PORT link" (STD p. 3.3-12)

(2) the thing assigned

e.g., "assigned PLA" (PPS 3.4.2.2.2.3.2)

"unassigned PLA"

Not even defined in glossary to PPS (Appendix B)

UU (User Unit)
MU (Master Unit) } need to be of same data type

Level - the MU also has a level

Support - Used to mean:

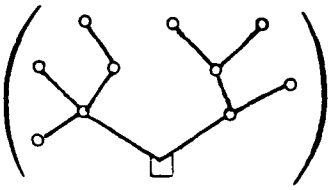
U-supp? (u_1,u_2)

T-supp? (t_1,t_2)

T-supp? $(ALP(u_1),t_2)$ "PLA is supported by a UU..."

Cooperate - not defined in glossary

Tree - doesn't mean links and nodes
means sets of time slots



Frame - glossary definition doesn't mention it
is composed of time slots

Rate - they mean period

PORT_link - should be PLA or "t"

other - needs to be an operation

PLA state - simply means high or low period

Figure 4.3-1

TECHNICAL TERMS NEEDED FOR THIS SECTION

3.4.2.2.2.3 PORT link assignment selection. PORT link assignment selection shall, unless requested otherwise, determine the best match between UU desired PL states and available PLAs. Available PORT link assignments shall be determined from the unassigned PLAs directly supported by active UUs. When no PLA is available for assignment to a UU designated for PL state change, that UU shall be requested to demand entry. The PLA selection criterion and PLA restrictions are defined in the following subparagraphs.

3.4.2.2.2.3.1 PLA selection criterion. The best match between the desired PL state and the available PLAs shall be selected using the following order:

a. PLA with an exact match between the desired and available rate (period)

b. PLA where the desired rate is lower than the available rate

c. PLA where the available rate is lower than the desired rate

For otherwise identical PLAs, selection shall be made in the following preference order:

d. PLA with lower levels

e. PLA with a closer rate match to the desired rate

f. PLA with earlier start frames.

3.4.2.2.2.3.2 PORT link restrictions. Available PLAs shall not be considered for assignment if the PLA either is supported by a UU that already cooperates in a PLA with the specified UU, or is active in frames that coincide with the specified UU's other assigned PLA. Whenever the MU is the supporting unit of an available PLA, all unassigned A-level PLAs shall be considered.

Each PLA shall be supported by only one UU. No two UUs shall have the same PLA. A PLA previously assigned to a UU shall not be available for reassignment until its deassignment or replacement has been explicitly acknowledged by:

a. A PLA command acknowledge

b. A mode command acknowledge

c. UU time out.

The A-level PLAs shall be assigned to different trees until all allocated trees have been used at least once.

3-43

Figure 4.3-2

OPERATIONS AND FUNCTIONS IN THIS SECTION

compare desired period with period of t_{old}.

3.4.2.2.2.3 PORT link assignment selection. PORT link assignment selection shall, unless requested otherwise, determine the best match between UU desired PL states and available PLAs. Available PORT link assignments shall be determined from the unassigned PLAs directly supported by active UUs. When no PLA is available for assignment to a UU designated for PL state change, that UU shall be requested to demand entry. The PLA selection criterion and PLA restrictions are defined in the following subparagraphs.

should be in different section?

3.4.2.2.2.3.1 PLA selection criterion. The best match between the desired PL state and the available PLAs shall be selected using the following order:

a. PLA with an exact match between the desired and available rate (period)

b. PLA where the desired rate is lower than the available rate

c. PLA where the available rate is lower than the desired rate

For otherwise identical PLAs, selection shall be made in the following preference order:

d. PLA with lower levels

e. PLA with a closer rate match to the desired rate

f. PLA with earlier start frames.

g. *missing!*

δ_ε δ_β δ_γ

3.4.2.2.2.3.2 PORT link restrictions. Available PLAs shall not be considered for assignment if the PLA either[is supported by a UU that already cooperates in a PLA with the specified UU,]or[is active in frames that coincide with the specified UU's other assigned PLA.] Whenever the MU is the supporting unit of an available PLA, all unassigned A-level PLAs shall be considered.

δ_4 *not necessary; t_{μ}*

meaningless or false

[Each PLA shall be supported by only one UU.] [No two UUs shall have the same PLA.] A PLA previously assigned to a UU shall not be available for reassignment until its deassignment or replacement has been explicitly acknowledged by:

a. A PLA command acknowledge

b. A mode command acknowledge

c. UU time out.

doesn't belong in this section

The A-level PLAs shall be assigned to different trees until all allocated trees have been used at least once.

δ_α

3-43

Figure 4.3-3

where their verbal descriptions appear in the PPS test; compare to the control map). (2) Some statements are either meaningless or false, e.g., "Each PLA shall be supported by only one UU." First of all, UUs U-support other UUs and Logical Times T-support other Logical Times, so a UU can't support a Logical Time (and certainly not a "PLA"). We then might ask if it means "a Logical Time has only one UU assigned to it" (in which case it is trivally true, by definition), or if it means "a Logical Time is T-supported by only one other UU's Logical Time," in which case it is false, since any Logical Time is supported by several other Logical Times and we must assume they mean directly T-supported. But this is inherent in the way Logical Times are defined (in the data-type definitions). So it is not clear this has anything to do with the search algorithm. (3) Some statements are unnecessary. The statement "No two UUs shall have the same PLA" is redundant, since one can only make PORT Link Assignments from unassigned Logical Times in the first place. (4) Some statements properly belong in other sections, e.g., the statement which begins "A PLA previously assigned..." and which includes the three categories (a-c) has nothing to do with PLA selection, but, as we can see from the control map, tells which Logical Times are going to be in $T_{Unassigned}$. But this is determined by the result of applying the operation ALP(t) and the result will depend on which PLAs are currently in force. But if the CLEAN-UP function does actual assignment, then the timing of when we deassign PLAs properly belongs in that module, not as part of the search algorithm. Similarly, the statement "When no PLA is available for assignment to a UU... that UU shall be requested to demand entry" is properly the response, or recovery from REJECT in FIND-PLA, and although it could conceivably be generated by the top level of FIND-PLA (i.e., generate a DEMAND-ENTRY-REQUEST), again this seems properly the function of a module outside FIND-PLA, as noted in Footnote #4.

To reiterate, this sample specification for Section 3.4.2.2.2.3 [11], Figure 4.3-4, has, of course, not solved all of the problems, some of which require more system-wide solutions. But it has pointed out and suggested possible resolutions, which might be taken into consideration as the specification of the PLRS Network Manager continues.

SAMPLE RE-WRITE OF PPS SECTION 3.4.2.2.2.3

3.4.2.2.2.3 PORT LINK ASSIGNMENT SELECTION

This module attempts to find a logical time whose period is as close as possible to the period generated by the PORT Link State Transition Processing module, and passes this new logical time to the PORT Link Assignment and Correction module.

	VARIABLE	DATA-TYPE	MEANING
INPUTS:	P	integer	Recommended period
	u_i	User Unit	The UU being considered for reassignment
	t_{old}	logical time	The LT of the PLA being considered for change
	T	tuple (of logical time)	The possible logical times*
OUTPUTS:	t_{new}	logical time	The new LT found
	OI message MTC message	data-type to be specified	

*This could be list, file, array, depending upon implementation layer considerations.

The module first conducts a Search for a new logical time, then detects, if necessary, a failure to find one, and responds with error message.

First the function f_0 (Figures 4.2.1-3 and 4.2.3-1) computes (or calls from memory) the logical times which are unassigned: ALP(t) = REJECT. This function outputs the set (list, array, etc.) $T_{Unassigned}$.

The next function, f_1, finds which are the logical times that T_support the logical times in $T_{Unassigned}$. ($f_1 = \hat{T}_{supporting} = DTsupp[T_{Unassigned}]$))

(The process in functions f_1 - f_4 is illustrated schematically in Figure 4.2.1-5.

Figure 4.3-4

We then find which of these logical times are PORT Link assigned to the user unit u_i' communicants. (f_2: T_{C_i} IS FORMED FROM $\hat{T}_{supporting}$ FORALL Element?(ALP(t),C_i))

We then find which logical times the communicants' logical times support (f_3) and divide them into A-level and non-A-level logical times (f_4).

We can now conduct the search for the best match, after first eliminating the logical times which fail to meet certain requirements.

For the A-level ones, T_A, we attempt to have at least one assignment for every tree as soon as possible in the beginning stages of building the network; therefore we apply f_α, which checks to see if there are any "empty" trees, and if so, picks a logical time from one of them if possible (Figure 4.2.3.2-1).

The rest of the tests, f_β- f_Δ, are run first on the A-level logical times, and if no candidate for PORT Link Assignment is found, the same tests, f - are run on the rest of the logical times, T_{C_i}-A.

Function f_γ checks the logical times to make sure the PORT paths formed by such an assignment will be district; i.e., not cross or form loops, either real (in the same tree) or virtual (in different trees) (Figure 4.2.3.-1).

Function f_4 eliminates from consideration those logical times which are active in the same frames as the User Unit u_i's other assigned logical time (Figure 4.2.3-1).

The final operation, f_Δ, then attempts to find a logical time whose period matches most closely the given period (Figure 4.2.3.4-1), and if more than one is found, breaks the tie using the criteria indicated in the operation (Tie Break, Figure 4.2.3.4-2 and Figure 4.2.3.4-3).

If at any point the set of logical times being considered becomes empty, a REJECT is generated and passed along the top levels of the various functions and is output as the value of t'_{new}. If t'_{new} = REJECT, then t_{new} is set equal to t_{old}, and appropriate meassages to the I/O and MTC are generated (Fig. 4.2.1-2).

Figure 4.3-4 (con't)

5.0 REVIEW OF SUBSTANTIVE PROBLEMS IN REAL TIME PLRS DESIGN

In the beginning of the PLRS project, we reviewed the preliminary Program Performance Specification [ ]. It has been the case that many of the errors we detected in this primary document have been cleared up in the final PPS. In addition, many of the things which we found missing, ambiguous, or wrong in the PPS have been straightened out in the STD and through discussion with members of the PLRS project. It seems as if the closer the system gets to implementation, the better the system is specified. (An example of this is provided by the PORT Link Assignment Command in the STD, which specifies the necessary data for a PLA better than the entire discussion in the PPS.) While the trend toward better design as the system moves closer to implementation is encouraging, it is not the best way to develop systems according to HOS principles. And, despite the improvement, some substantial errors remain. This section will describe how these problems and errors are detected. A representative list of them will be described in some detail. Finally, we will summarize them by category and discuss how the consistent application of HOS principles can be used to avoid them.

5.1 Examples of Problem Categories

5.1.1 Confusion between the Network of Units and the Network of Logical Times

This confusion is displayed in several ambiguities. For example, the term tree is used both in the conventional graph sense (as a hierarchical set of units) and in a special sense of a set of timeslots. We will describe three problems arising out of this confusion.

PLA Confusion

Throughout the PPS, there is confusion regarding PORT Link Assignments (i.e., a set of transaction groups - 16 timeslots - during which another set of PUs are communicating) and PLAs, which are often used in the sense of possible or potential transaction groups. This confusion is reflected, for example, in the concept of "unassigned PLAs" (see PPS, Sect. 3.4.2.2.2.3). The fact that PORT Link Assignment is not a glossary entry is not helpful.

The implication of this confusion is that the procedures for finding available transaction groups to be used as a PLA cannot be clearly specified.

This problem was identified by trying to determine formal data types to be used in the control map. (See discussion in Section 3.6 and 3.7.)

Confusion between Units and Logical Times

Clearly, information regarding both units (e.g., communicants) and Logical Times (e.g., which contain cycle rates) must be considered in managing the network. However, these two concepts must not be confused. For example, when (in PPS Sect. 3.4.2.2.2.3.2) it is said, "Each PLA shall be supported by one only UU", the concepts are mixed. Elsewhere (PPS Sect. 3.4.2.2.2.1.3) PLA's support other PLAs and units support units.

This problem was discovered while formulating the data types.

Primitive Operations

In the PPS (especially Section 3.4.2.2.2.3.2), the confusion between links and units leads to a corresponding confusion between the primitive operations which might be performed on each. In the statement, "Each PLA shall be supported by one one UU", the implication is that UUs support PORT Link Assignments (rather than UUs supporting UUs). As stated the sentence suggests that "find the unique UU supporting this PLA" would be a legitimate operation. In actuality, the appropriate operation is: "find a UU among the set of communicants whose PLA may support another PLA."

Implications of the confusion between primitive operations is that the software developers cannot be certain of quite what qualities are significant, hence they cannot be sure of how to implement the constraints.

This confusion was discovered during the preparation of the operations for use in the control map.

5.1.2 Potential Sources of Loss-of-Control

As the system is described in the PPS, there is not strict continuity in the flow of control in the Network Manager's operations. For example (PPS 3.4.2.2.1.1.3), Handover-in UU processing is described in terms of what will have happened after the process is completed. Working from the available description, we can assume that MTC informs NM that a UU is to be handed-over. (Network Control marks this UU as "zero rate" and issues a tree allocation command.) MTC passes this command onto the UU. Then the PPS says, "Upon acknowledgement of this command, network control shall generate a handover-in successful notice." As written, this description does not include the possibility that the UU fails to respond. If this were to happen, it is not clear where the control stopped. Is the next function to be performed in network control, in message traffic control, in both, or in neither?

The serious implications of this sort of ambiguity are obvious! And it may arise, in basically the same form, for a wide variety of commands which expect a response from the UU community.

These problems were discovered by trying to build the control map of the network manager.

5.1.3 Inconsistent I/O Interfaces

In the original PPS, this sort of problem was very prevalent. The revised PPS has cleared up some of these problems, but most remain.

Name Changes

Several data elements have different names and different membership at different levels of documentation detail. For example, in Figure 3.3.5-1 [11], the OI inputs UU CONTROL to the NM; in Figure 3.4.2-1 [11], it inputs UU MODE CONTROL; in Section 3.4.2.1.2.4 [11], "the new NM function shall accept inputs from the OI function consisting of:... 1. Clear, 2. Passive, 3., Reenter, 4. Restart". While this may seem like a trivial problem, it must be resolved before the system can be expected to operate. Why not avoid the problem by a disciplined consistency from the very beginning?

Other examples of this kind of problem include missing UU OCCUPANCY, UU STATE, CURRENT CLA, CURRENT PLA, FORCE PLA ALERT, ZERO ALERT (or ZERO LINK ALERT), etc., in Figure 3.3.5-1 [11]; unique NETWORK VALUE, and others.

These inconsistencies were discovered while formulating the preliminary control map.

Identity of Data Items

This problem involves the completeness and the consistency of the PPS as a document from which to develop software. Therefore, it should be self-contained. But it is not clear within the PPS what UU MODE (for example) signifies. Similarly, the meaning of R and S-type PORT Link rates is unclear. The fact that the identity of data items seems to change according to documentation level was discussed above.

The implications of all these inconsistencies and ambiguities is that unnecessary confusion is created in the minds of the software developers; this confusion increases the likelihood of coding error.

These problems were discovered by trying to formally specify the necessary data types for the HOS control map, in which specificity and consistency are required.

5.1.4 Functions Not Well Defined

In several places in the PPS, various functions are invoked (or operations implied) which are not consistent with prior definition of data items or other functions. In the software development process currently being used for the PLRS project, these problems do not arise until the implementation stage. But in the software engineering process implied by the HOS methodology, they are encountered early, and, therefore, can be resolved before extreme committment of time and money are made. Furthermore, with precise preliminary specifications, some problems do not arise at all.

PLA Two-Valued on Unit

This is simply the problem that each unit has two PLAs. In the implied search procedures to discover which PLA allows a UU to receive commands from the MU, two outputs are possible. Everywhere in the PPS, whenever a call for a PLA associated with a given unit is initiated, it is assumed that one is found. Nowhere in the PPS is it suggested which one.

The implications of this ill-defined function (any function with two possible outputs for a given input is necessarily ill-defined) are that the choice will be made and it properly is a choice to be made by an integrative systems designer with ample perspective on its effects; not one to be made by a staff programmer for possibly idiosyncratic reasons.

This implied two-valued function was discovered when we tried to make the control functional.

Cooperate Ill-Defined

The critical concept of "cooperate" is used in three different senses. In PPS 3.4.2.2.2.3.2, the phrase, "the PLA either is supported by a UU that already cooperates in a PLA with the specified UU," implies that the cooperating UU is on the same PORT Path (and at a lower level) than the specified UU. Elsewhere (in the case of CLAs), cooperating UUs are those who "listen" for a specified UU. But in the glossary, a cooperating unit is defined as "A PLRS net member whose transmissions are received by another unit." This definition makes all communicants "cooperating units"!

The implication of using one term to signify many concepts is that different people might make conflicting judgements about its meaning. If the resulting implementation employed a test called COOPERATE(UU_i,UU_j), it would be expected to yield different values in different modules.

This problem was discovered when the above test was considered for the control map.

Period = $\frac{256}{R}$, hence Zero Rate = Period $\rightarrow \infty$

Zero rate has an intuitively plausible meaning--namely the UU never automatically relays or listens for transmissions. However, a problem arises in the representation of zero rate UUs, because the parallel

concept of period is also often used to characterize them. Clearly, Period=$\frac{256}{R}$ for all rates allowed between 1 and 256. But when the operations implied by the PPS are attempted on zero rate UUs, period is transformed to "infinity." This problem can be handled at the implementation stage by a minor check for divide-by-zero or the provision of some special code for "infinity," but it is one more thing that could go wrong if not noticed and treated.

The implication of this minor problem is the possibility that the implied arithmetic might actually be attempted. Presumably, this would result in a floating-point check and system halt.

This problem was detected when we tried to specify data types for cycle rate and period.

5.1.5 Incomplete or Wrong Algorithms

Although the PPS is not consistently at the level of detail of operational algorithms in those cases where the criteria for some system action is important (e.g. chose a potential PLA, declare a link unreliable and a candidate for replacement, etc.), the PPS should specify either all the criteria (complete the algorithm) or state that the algorithm is incomplete. There are several cases in which the PPS seems to specify how a system decision is to be made, but omits crucial detail.

Tie-Breaking Not Complete

Section 3.4.2.2.2.3.1 specifies several criteria for the selection of a PLA to a unit when one of its two desired PLAs is missing. But the criteria as stated are not sufficient to determina a unique PLA. That is, after applying all the constraints, several possible PLAs would often remain. How is the PLA Control function to select from these candidates?

The effect of this oversight is to postpone the decision to implementation time. Eventually, the choice must be made. It ought to be made explicitly and on a rational basis and not left to programmer whim.

This omission was discovered when the function, "Find PLA," was specified in control map terms.

Inflexibility of PLA Rate Mix

In Sections 3.4.2.2.2.3.1 and .2, the PPS specifies the PLA selection criteria. Because reliability is a PLRS system goal, one restriction on the two PLAs that serve a given UU is that they are relayed through no common UUs. This is stated as: A PLA shall not be considered if it "is active in frames that coincide with the specified UUs other assigned PLA." The selection criteria of Section 3.4.2.2.2.3.1 are not sufficient to ensure this.

The implication of this is that the module which performs the "FIND-PLA" function will not have all the information it needs, unless this problem is cleared up in implementation.

This deficienty was discovered while formulating the "FIND-PLA" function in the control map. See discussion in Section 3.4.3.

Can't Find What Transaction Groups Are Available with Respect to Time

Again, in the FIND-PLA function, it is necessary to determine which transaction group ("unassigned PLA's" in PPS terms) are candidates for forming a PORT Path to a given UU. One criterion is based on UU data, namely the set of communicants. The other criteria include those in Sections 3.4.2.2.2.3.1-2 [11]. But nowhere in the PPS is it specified how the available transaction groups can be discovered. We have solved this problem by using the control map and primitive operations on the HOS data types, but feel that the PPS should have made it explicit that this algorithm was incomplete.

The implications are that there are several alternative ways to go about this necessary function. Without noting that the choice of method is being left to a later stage, a poor alternative may be implemented without opportunity for review.

This omission was discovered by developing explicit procedures for "FIND-PLA" in the control map. See Section 4.2.3.4 for a more detailed explanation.

What is "PORT Link Time Assignment"?

In the Tree Allocation Command, provision is made for two 3-bit fields which are called PORT Link Time Assignments. We found it useful to refer to the actual commands because they are so much more explicit (in most cases) than the discussion in the PPS. We cannot find a specification of this data element anywhere, however.

The implication of this omission is simply to leave uncertainty in the specification of quite what this command is meant to do.

This omission was discovered by inspection of the STD in the process of formatting data types.

Error Detection and Recovery

Many places in the PPS, an operation is initiated which may or may not be successful (produce the desired effect). The most typical case is the generation of a command to a UU to take a Logical Time assignment or to change mode. The UU may or may not respond. Usually, the specified action is to generate a "zero alert." This in itself is not sufficient recovery procedure. Because the action to be taken subsequent to the generation of the alert is unspecified, there is no guarantee that the system will ever get back on track.

Error detection and recovery is sufficiently important that it ought to be explicitly incorporated early in the program development process. A single missing recovery procedure could have disasterous consequences in the field.

These considerations are an explicit part of the HOS philosophy and follow directly from the principles regarding completeness of control.

5.1.6 Overall System Considerations

This section discusses some topics which we feel are important considerations in the Network Manager function and (probably) for the system as a whole. They are not problems in the sense of the above, but perhaps they should have been treated in the PPS.

Redundant Operations

It seems as if some of the functions and operations necessary to implementthe Network Manager specification can be used unchanged in several places. For example, specification for PLA is identical to CLA. Both are assignments of Logical Times to UUs, but the algorithms for finding them may be different. Also, zero alerts are generated in several places. The opportunity to exploit identical code for redundant operations can be seized only if it is recognized early. In the formulation specified in our control map, for example, PLAs and CLAs have exactly the same representation (although they are different data structures because they have different meanings). Similarly, some of the operations on them are identical in form. The control map building exercise is an excellent way of discovering such operations.

The Operator and the Users as Part of the System

The PPS fails to consider the MU operator and the community users as part of the system. This is an inadequacy because many of the control loops in the Network Manager (and other portions of the PLRS) are closed only through the operator or through the community. That is, the software will generate a condition (an operator message or a command to a UUO and issue it. The next time control passes back to the software, it is because the operator or a user has take some action. At a minimum, the range of operator actions should be related back to the software functions which (in some sense) initiated them.

Treating the PLRS as a closed system when it actually is open to both the operator and the users invites loss-of-control problems.

This situation became evident when we formulated the control map and found that many of the Network Manager's control loops were closed only through the operator.

5.2 Statistical Summary of Problems

CATEGORIES OF PROBLEMS FOUND	Implication[1]	Frequency[2]	Discovery[3]
CONFUSION BETWEEN LOGICAL TIMES AND UNITS			
PLA Confusion	Serious	Often	Formal Data Type
Units/Logical Times Confusion	Serious	Often	Control Map
Primitive Operations	Serious	Often	Control Map
POTENTIAL SOURCES FOR LOSS OF CONTROL			
Command/Acknowledge	Crucial	Often	Control Map
INCONSISTENT I/O INTERFACES			
Name Changes (Mode Control)	Minor	Often	Control Map
Identity of Data Items	Minor	Sometimes	Data Types
FUNCTIONS NOT WELL DEFINED			
PLA Two-Valued	Crucial	Once	Control Map
Cooperate Ill-Defined	Serious	Once	Control Map
Period=$\frac{256}{R}$, hence Zero rate $\rightarrow \infty$	Minor	Sometimes	Data Types
INCOMPLETE OR WRONG ALGORITHMS			
Tie-Breaking not Complete	Crucial	Once	Control Map
Inflexibility of PLA Rate Mix	Serious	Once	Control Map
Can't find what transaction groups available with respect to time	Serious	Once	Control Map
What is "PORT Link Time Assignment in Tree Allocation Command?	Serious	Once	Data Types
Error Detection and Recovery (various)	Crucial	Often	Control Map
OVERALL SYSTEM CONSIDERATIONS			
Redundant Operations (e.g., PLA+CLA)	Serious	Often	Control Map
Operator as Part of System	Serious	Sometimes	Control Map

[1] Implication means expected magnitude of consequences. It is composed of the probability that the problem would go undetected times the performance sacrifice if it occurred.

[2] Frequency means that it can occur from 2-10 times--often 11-50 instances.

[3] Discovery refers to the exercise that was being worked on when the problem was recognized.

5.3 Considerations for Software Verification

We are given a hint only of the procedures that Hughes employs to ensure that the software will work as designed. The major weakness of these procedures (sketched in the Hughes Design Plan), from the perspective of HOS techniques, is that they are informal. That is, they depend on the subjective judgement of the software creators as to possible failure modes.

Now, this practice often may be adequate in the sense that the designer may usually have a good appreciation of the environmental circumstances which will impact his code. But, occasionally, something totally unexpected might happen. As a purely speculative example, consider a spurious acknowledgement. Depending on the details of implementation (within the constraints of the PPS), a spurious acknowledgement could hang up the software--searching for the (never-issued) command that it thought was being acknowledged!

One result of this approach which may prove troublesome is the dependence of the designers of the Online Simulation Program on the designers of the Real Time PLRS. Because the simulation program is intended to demonstrate the MU's operational ability to meet specified requirements, it is important that it exercise all possible conditions. From the available documentation, it seems like one of the functions of the Real Time PLRS module designers is to suggest modes in which their modules ought to be exercised. Because the conditions they anticipate will be the ones they design for, the power of the simulation to detect error is reduced.

There is a better way to ensure software validity! Because a control map constructed along the formal axioms of HOS must incorporate all possible cases (values, inputs, etc.), it can be used to generate an exhaustive set of test conditions for each functional module. Invalid data cannot hang the program--in the worst case, they can only propagate REJECTS to the highest level, exactly the desired outcome.

Software engineering directed by formal rules is far more secure than that which depends merely on multiple levels of review.

FOOTNOTES

Footnote 1

An object or name can be assigned or unassigned, but an assignment can hardly be assigned or unassigned. Here it is time intervals which are either assigned to UUs or not assigned to UUs, and the match of a particular UU, i.e., u_i with a particular set of time intervals in which it is active, i.e., t_j, is what constitutes the PORT Link Assignment: (u_i,t_j)--not the t_j alone.

Footnote 2

It should be noted that there seems to be an inconsistency here in the PPS text. Although the PORT Link Assignment command format indicates that the Logical Times assigned to UUs are being specified using the exponent on the Period, throughout the statement of various algorithms in the text this is referred to as Cycle Rate, rather than period. We understand Cycle Rate to be equal to 256/Period, so that it would be a simple matter in various modules to convert from one to the other, except for the fact that Cycle Rate is often set to zero as a means of indicating a necessary change in PLA (c.f. PPS Section 3.4.2.2.1.1.2, p. 3-38, "Reentry UUs shall be designated as being in a zero rate PORT link state and shall be process by PLA control."), and since P = 256/CR, if CR = 0, P = ∞,not one of the periods allowed for. There will have to be some sort of ad hoc patch; we simply note the inconsistency here.

Footnote 3

Given, for example, in the STD, Section 3.5.5, p. 3.5-10/12, "Advantages of Network Redundancy."

Footnote 4

Note Axiom 6b is further complicated by taking into account the Master Unit, which we must do for consistency, since the MU does support the other UUs. For example, if we ask what unit directly supports some unit, the answer (if it is A-level) may be the MU. Furthermore, may we not take the alternative of using Axiom X , Usupp?(m,u) = True, if there are multiple MUs, also if we consider new entry UUs, which may be communicants without being supported? This requires further study, beyond the scope of this report. The final formulation of the axioms would presumably make a choice between Axiom X and Axiom 6b.

Footnote 5

We picture a 16-frame epoch rather than a real 256-frame epoch for simplicity.

Footnote 6

CROSS Link Assignments are omitted from this discussion, although they would be handled in an analogous manner. In addition, for the sake of simplicity, the behavior (specification) of the Master Unit is omitted from the following discussion.

Footnote 7

Additional operations and axioms for data type Logical Time suggested by Stephen Kenton.

Footnote 8

Including the error recovery as part of FIND-PLA was done primarily for pedagogical purposes: to show how an error message might be incorporated into the specification. In a complete specification for the PLA Control submodule, however, the error detection and recovery would be done at a higher level. That is FIND-PLA would consist basically of what is called SEARCH in Figure 4.2.1-3a. This might generate a REJECT as the value of t'_{new} which would be passed up to the operator system.

Footnote 9

That is, if $(t_{OLD},t_{\alpha}) = PLA(u_i)$, where t_{α} is u_i's other assigned Logical Time, or stated inversely, if $u_i = ALP(t_{OLD})$, then the duple (u_i,t_{OLD}) must be stored in memory somewhere; this would be replaced by the duple (u_i,t_{NEW}). That is, the CLEAN-UP function would do the actual changing of PORT Link Assignments; $PLA_1 \rightarrow PLA_2$ or rather $(u_i,t_1) \rightarrow (u_i,t_2)$.

Footnote 10

It may be instructive to the reader to work through the recursion in SET-TEST. If so, note that the final REJECT obtained is not an error; it simply marks the end of the ordered set (list, etc.). For example, if we are examining the items in the list $T_0 = (7,3,8,2)$ and only 3 and 2 happen to meet the required criterion being tested, then the following action occurs:

T_1 IS FORMED FROM T_0 FORALL t < 5;

INPUT: $T_1 = (7,3,8,2)$

REPLACE

(7,3,8,2) = 7 and (3,8,2); 7 is eliminated.
(3,8,2) = 3 and (8,2); 3 is saved.
(8,2) = 8 and (2); 8 is eliminated.
(2) = 2 and REJECT; 2 is saved.

2 and REJECT = (2)
3 and (2) = (3,2)

OUTPUT: $T_1 = (3,2)$

Thus as one can see, the REJECT is simply an end-marker, much like the element NIL in the LISP language.

Footnote 11

The operation Element?(x,S) is used here in the general sense. The AXES specification language has the operation defined so far only for the case in which S is a set; however, it is clear this could be extended to the cases where S is of some other set-like data type (e.g., lists, arrays, files, tuples, etc.).

REFERENCES

[1] Bridge, R.F. and Thompson, E.W. "A Module Interface Specification Language", Information Systems Research Laboratory, University of Texas at Austin, Technical Report 163, Dec. 1974.

[2] Cushing, S. "The Software Security Problem and How to Solve It", TR-6, Revision 1, Higher Order Software, Inc. (hereafter cited as HOS, Inc.), Cambridge, MA, July 1977.

[3] Guttag, J., et al. "The Design of Data Structure Specifications", Proceedings of 2nd International Conference on Software Engineering, Oct 1976.

[4] Hamilton, M. and Zeldin, S. "Higher Order Software--A Methodology for Defining Software", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976.

[5] Hamilton, M. and Zeldin, S. "The Manager as an Abstract Systems Engineer", Digest of Papers, Fall COMPCON 77 (Washington, D.C.), IEEE Computer Society, Cat. No. 77CH1258-3C, Sept. 6-9, 1977.

[6] Hamilton, M. and Zeldin, S. "Verification of an Axiomatic Requirements Specification", A Collection of Technical Papers, AIAA/NASA/IEEE/ACM Computers in Aerospace Conference (Los Angeles, CA), Oct. 31-Nov. 2, 1977.

[7] Harel, D. and Pankiewicz, R. "A Universal Flowcharter," TR-11. HOS, Inc., Cambridge, MA, Nov. 1977.

[8] Heath, W. "Some Specifications for the Operating System of the Apollo Guidance Computer (AGC)" in "Techniques for Operating System Machines", TR-7, HOS, Inc., Cambridge, MA, July 1977.

[9] IBM. "HIPO: Design Aid and Demonstration Tool", IBM, SRZO-9413-0, 1973.

[10] Jackson, M.A. Principles of Program Design. Academic Press, NY, 1975.

[11] "Program Performance Specification for Real Time PLRS Program--Vol. 1", Revision Original (Final), Hughes Aircraft Co., Ground Systems Group, Fullerton, CA, July 29, 1977.

[12] Robinson, L. and Holt, R.C. "Formal Specifications for Solutions to Synchronization Problems", Computer Science Group, Stanford Research Institute, 1975.

[13] Ross, D. "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, Jan. 1977.

[14] "System Technical Description", FR 77-14-90, Hughes Aircraft Company, Ground Systems Group, Fullerton, CA, 9 February 1977.

[15] Teichroew D. and Jershey, E.A. III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, Jan. 1977.

[16] Thiersch, C. "Observations of the Network Manager with the Use of Control Maps", PLRS Memo #9, HOS, Inc., Cambridge, MA, Sept. 2, 1977.

Appendix I

Preliminaries of HOS

Appendix II

Preliminaries of AXES

APPENDIX I*

Preliminaries of HOS

In HOS, the decomposition process for a system results in a tree structure. At the start of the decomposition process, the entire system is represented by the root of the tree which, hopefully, represents the requirements for the system. This definition, however, has many implicit (hidden) requirements. In order to arrive explicitly at the complete definition of the system, the root is decomposed by replacing it with a *nodal family* (a particular parent node and all of its offspring), which represents the decomposition of the root. This decomposition process, that of replacing a function by its nodal family, can be continued until the entire system has been specified. The resulting tree represents the complete system specification, where the leaves represent primitive operations on the data types represented by the variables at those leaves. It may turn out that during the decomposition process a requirement is shown to be erroneous or missing. In such a case, an iteration of the system description is required.

The parent node of the nodal family controls its offspring. When referring to this control relationship, the parent node will be called a *module*, and its offspring will be called *functions*] The offspring of the nodal family are the functions required to perform the *module's corresponding function* (MCF) (i.e., the function that the nodal family replaces).

In the sections that follow, the variable that represents the domain elements of a function is referred to as the *input variable*, and the variable that represents the range elements of a function is referred to as the *output variable*. Individual domain and range elements may be called inputs and outputs, respectively.

A module, in performing its corresponding function (Figure AI-1), is responsible for determining if the inputs received are in the intended domain of the MCF. If an input is not in the intended domain of the MCF, it is in the unintended domain of the MCF and maps to a special value which is a value of every data type, the value Reject.

In a sense, the improper input element is not in the domain of the module's corresponding intended function (MIF), but is in the domain of the MCF, i.e., the module's corresponding unintended function (MUF).

Properties of the Primitive Control Structures

While a function can be decomposed in many ways, the HOS axioms [2] provide rules for the construction of nodal families (i.e., the decomposition of a function). From these axioms, three primitive control structures, which are used for functional decomposition, are derived [1].

These control structures are: *composition*, *set partition*, and *class partition*.

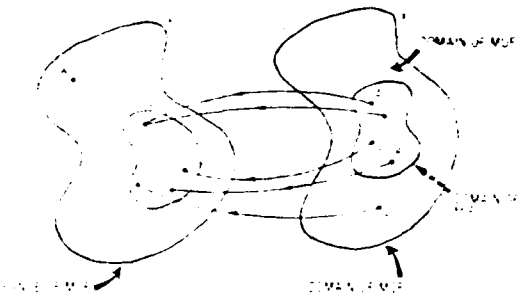


Figure AI-1. Illustration of a Function from X into Y

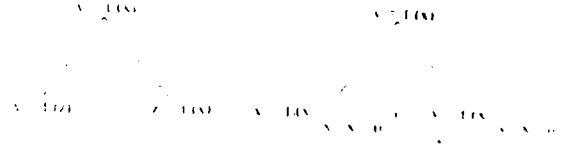


Figure AI-2.
An Example of Composition

Figure AI-3.
An Example of Set Partition

Composition is illustrated in Figure AI-2. In order to perform $f_1(x)$, the function f_2 must first be applied to x which results in output z. z then becomes an input to f_1 which produces the desired range element of the overall function.

It is important to observe the following characteristics of composition (characteristics are explained with respect to the example in Figure AI-2):

(1) One and only one offspring (specifically f_2 in this example) receives access rights to the input data, x, from module f_1.

(2) One and only one offspring (specifically f_1 in this example) has access rights to deliver the output data, y, for module f

(3) All other input and output data that will be produced by offspring controlled by f_1 will reside in *local* variables (specifically "z" in this example). Local variable, "z", provides communication between the offspring f_2 and f_1.

(4) Every offspring is specified to be invoked once and only once in each process of performing its parent's MCF.

(5) Every local variable must exist both as an input variable for for one and only one function and as an output variable for one and only one different function on the same level.

Set partition, which involves partitioning of the domain, is illustrated in Figure AI-3. In the example, the set which comprises the domain is partitioned** into two subsets. For set partition, only one of the offspring will be invoked for each performance of the MCF at f_1 (the determination being based on the value of "x" received) and that offspring will produce the required range element for its parent module when it is performing.

The following characteristics with respect to set partition should be observed:

(1) Every offspring of the module at f_1 is granted permission to produce output values of "y".

(2) All offspring of the module at f_2 are granted permission to receive input values from the variable "x".

*Excerpted from [3].

**Partitioning implies the subdivision of the original set into non-overlapping (i.e., mutually exclusive subsets)

(3) Only one offspring is specified to be invoked per input value received for each process of performing its parent's MCF.

(4) The values represented by the input variables of an offspring's function comprise a proper subset of the domain of the function of the parent module.

(5) There is no communication between offspring.

Class partition is illustrated in Figure AI-4. While set partition involves partition of the domain into subsets, class partition involves partition of the domain variables into classes and the partition of the range variables into classes. In the example, it is assumed that the domain variable has an associated data structure comprised of two parts, "x_1" and "x_2". Likewise, the range variable has an associated data structure with the same number of classes as the domain's data structure.

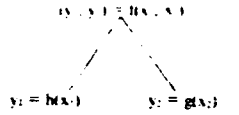


Figure AI-4. An Example of Class Partition

The following characteristics with respect to class partition should be observed:

(1) All offspring of the module at f are granted permission to receive input values taken from a partitioned variable in the set of the parent MCF domain variables, such that each offspring's set of input variables is non-overlapping and all the offspring's input variables collectively represent only its parent's MCF input variables.

(2) All offspring of the module at f are granted permission to produce output values for a partitioned variable in the set of the parent MCF range variables, such that each offspring's set of output variables is non-overlapping and all the offspring's set of output variables collectively represent the parent MCF variables.

(3) Each offspring is specified to be invoked per input value received for each process of performing its parent's MCF.

(4) There is no communication between offspring.

APPENDIX II*

Preliminaries of AXES

I. Axes Syntax Description

AXES is a formal notation for writing definitions of systems. These systems include systems which are mechanisms for defining other systems. Thus, for example, we could define a set of specification "macros" which collectively could form a language for defining a system or family of systems. Since each language statement would be a definition "macro" based on an integrated Higher Order Software (HOS) control hierarchy [2], the resource allocation to a particular machine could then be addressed independently from the definition of the system. Although it is not a programming language, AXES is a complete and well-defined language capable of being analyzed by a computer. AXES is intended to provide commonality between systems. Although users will have flexibility to choose different building blocks, these building blocks, when "compiled," will bring users to a common meeting ground with all other users of AXES.

AXES systems can be hierarchically decomposed into complete system specifications with the use of abstract control structures that relate members of algebraically defined data types. Three primitive control structures have been derived from six axioms that define completeness of control [1]. These primitive control structures provide rules for the definition of communication, parallel processing, and selection of functions. From a combination of primitives, we can form more abstract control structures (e.g., recursive functions).

The syntax of AXES [4] provides the mechanisms to specify control structures and data types. The purpose of AXES is to express a system specification which is equivalent to that same specification expressed graphically as a control map. Control structures can be described in AXES as structures, operations, and functions. Whereas a *structure* is a relation on a set of mappings, i.e., a set of tuples whose members are sets of ordered pairs, an *operation* is a set of mappings which stand in a particular relation. An operation results, mathematically, from taking particular mappings as the arguments (nodes) of a structure. By a *function*, we mean a set of mappings which stand in a particular relation for which particular variables have been chosen to represent their inputs and outputs. Whereas structures and operations can be described as purely mathematical constructs, a function is a hybrid, consisting of a mathematical construct and a linguistic construct, i.e., an assignment of particular names of inputs and outputs. Note that our use of "function" is slightly different from what is meant by "function" in mathematics. For the latter notion, we use the term "mapping" throughout this paper.

In AXES, a new data type can be defined simply in terms of the operations that are to be performed on the data [4]. That is, a data type is defined algebraically rather than operationally by making true statements (or axioms) about the equality of two control structures in which all the nodes are operations. Each such control structure is defined in terms of primitive operations of the data type of interest or of previously characterized primitive operations of another type (previously characterized primitive operations include universal primitive operations that have been defined, each of which is associated with any member of any data type).

The axioms associated with the definition of a data type are only those we *need* to characterize the data type. There are, of course, other operations that we find useful for other purposes. We are free to define any operation we want on an already-defined type as long as the operation definition is consistent with the axioms of the type. A new operation can be characterized either as an *OPERATION* or as a *DERIVED OPERATION*.

In AXES, we specify the behavior of an operation without specifying its decomposition by writing it as a derived operation, i.e., by means of true statements that describe the behavior of the operation with respect to other already-defined operations. Either kind of operation could be written as a control map, if desired. They differ in how they are specified, not in what they are. What distinguishes both of these kinds of operations from primitive operations on their data type is that their existence is provable mathematically from the existence of the primitive operations and the axioms of the type. In fact, if an *OPERATION* (which defines a function) and a *DERIVED OPERATION* (which defines the behavior) are both used to define the same function, the behavioral properties can be checked against the refinement properties to prove the correctness of a definition.

In describing AXES we will use variables and constants themselves to make statements about the values they name, and we will use the names of variables and constants to make statements about the variables and constants themselves.

To differentiate an object from its name, we introduced the "use-mention distinction" [7] in AXES. That is, we can talk about an object only by using a name of the object. (To talk about a man, for example, we have to use a sentence that contains the man's name, not the man himself.) The notation conventionally used for this is *enclosure within quotation marks*. To form the name of a given name (or written symbol of any kind), we include that name (or symbol) in quotation marks. (Successive embedding of quotation marks can be used if we want to talk about names, names of names, and names of names of names.)

In AXES, a *constant* symbol is the name of a particular value

*Excerpted from [6]

and corresponds to a proper noun like "John." A *variable* is the name of more than one possible value and corresponds to a common noun like "a man."

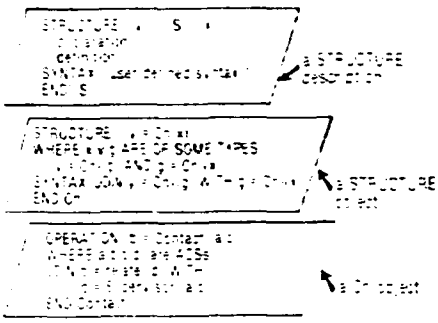


Figure AII-1. An Example of Abstract-Control-Structure Definition Layers with Respect to AXES

For example, in Figure AII-1, the top-most box is a description of part of AXES itself. The top-most box describes the AXES objects required to define a *STRUCTURE* in AXES. The sentence

"*STRUCTURE*:" y "=" S "("x");"

makes a statement about values by using the variable "y", "s", and "x" and about constant symbols by using the quotation-marked symbols such as "'*STRUCTURE*'", and "'='".

The middle box encloses an AXES object itself; that is, the middle box encloses the definition of a language statement derived from the definition of an AXES module. The Composition (Cn) *STRUCTURE*, defined in Figure AII-1, is one of the three HOS primitive control structures. Each primitive control structure has been defined as a *STRUCTURE* with AXES [4]*.

The middle box encloses an instance of the layer that the topmost box represents. If we could describe all of the structures that could possibly exist, then the complete set of structures would be the layer that the top-most box describes.

When a *STRUCTURE* in AXES is defined, the designer supplies the syntax (or description) so that a user of that structure can describe particular mappings that stand in the relation. For example, the bottom-most box in Figure AII-1 encloses an AXES object that is an instance of the Cn structure described in the middle box of Figure AII-1; that is, the bottom-most box is the definition of a system derived from the definition of a language statement, derived from the definition of an AXES module. In the bottom-most box, "b = Relate(d)" describes a particular function that "y = Cn (g)" represents in the middle box. Likewise, "d = Supervise (a)" represents an instance of "g = Cn(x)". In the middle box, the objects that "y", "g" and "x" represent are described in the statement

Where x,y,g *are of some types*;

This statement means that x, y, and g are variables whose values are of an unspecified data type. In the bottom-most box the *WHERE* statement is used to specify a particular data type, and the operation, Contact, is a particular mapping.

Other control structures can be derived from already defined control structures and operations that operate on variables of any type. Operations that operate on variables of any type are called *universal operations*. Primitive universal operations are defined as

$$x = \text{Clone}_1(x) \quad (1)$$

$$(x,x) = \text{Clone}_2(x) \quad (2)$$

$$con = K_{con}(x) \quad (3)$$

$$x_1 = id_1^2(x_1,x_2) \quad (4)$$

$$x_2 = id_2^2(x_1,x_2) \quad (5)$$

$$(x_1,x_2) = St(x) \quad (6)$$

$$x = T(x_1,x_2) \quad (7)$$

(1) and (2) are used to specify more than one variable with the same value. (3) is used to choose a constant symbol. (4) and (5) are used to select the value of one of a set of variables. (6) and (7) are related by T(St(x)) = x. These are used to create a value of a data structure from a value of a data type (i.e., St) or to create a value of a data type from a value of data structure (i.e., T).

Universal operations have, as their bottom nodes, universal primitive operations. Universal operations are defined as *STRUCTURE* in AXES because they operate on values which are variables.

We can now define a structure whose syntax can be used to define more than one system having access to the same value. Here, we use the universal operation

$y = id_a^b(x)$

which is used to select particular variables out of a set of variables as well as the universal primitive operation

$(x,x) = \text{Clone}_2(x)$

to determine the meaning of the relationship among the unspecified functions that appear as bottom nodes of the structure definition.

STRUCTURE: y = J(x);
Where y,g,w,h *are of some type*;
Where b *is a* NATURAL;
Where a *is a* SET (*of* NATURALS);
$y = J_1(g,w)$ *Join* $(g,w) = J_2(x)$;
$(g,w) = J_{12}(x,x)$ *Join* $(x,x) = \text{Clone}_2(x)$;
$g = J_{111}(x)$ *Include* w $id_a^b(x)$;
$g = J_{111}(h)$ *Join* h = $id_a^b(x)$;
SYNTAX: $y = J_1(g,w)$ *Cojoin* $g = J_{111}(h)$;
END J;

In using the syntax of a structure, an instance of the layer of the structure definition can be obtained. In the *Cojoin* structure, there are actually four unspecified mappings besides the top node: J_1, J_{111}, id_a^b, id_c^b. But in the use of the *Cojoin*, the value of "w" and "x" uniquely determines the particular id_a^b function. Likewise, the value of "h" and "x" uniquely determines the particular id_c^b. Thus, only J_1, J_{111} need appear in the syntax of the *Cojoin* definition. The collective set of values that replaces the variables described in the syntax can be traced to each node of the structure definition. For example, if

(a,b) = F(r,t,s)

is defined as

(a,b) = A(p,q,r) *Cojoin* (p,q) = B(r,t)

The first and second statement collectively form an instance of the *Cojoin* structure.

In this example,

"x" has the value "(r,t,s)"
"J" has the value "A"
"y" has the value "(a,b)"
"g" has the value "(p,q)"

*The syntax for a case partition is
$y = f_1(x)$ *Otherwise* $y = f_2(x)$,
Partitions of (x,y) *is an Partition*.
Here, the left subscript indicates a member of a member of a partition of "x".
The syntax for a class partition is
$y = f_1(x)$ *Include* $y = f_2(x)$.

"w" has the value "r"
"J_{111_2}" has the value "B"

and, since the input to F has three components, "b" in the structure definition has the value "3", since "w" has the value "r", which has one component, "a" in the structure definition has the value "I", and so on. The structure syntax names the objects necessary so that an instance of the structure definition is obtained. Any instance of a structure must itself be an HOS system.

In the *Cojoin* structure, systems that communicate with each other can access the same value. Likewise, we define other structures; one so that independent subfunctions can access the same value (the *Coinclude*), and one so that subfunctions whose invocation depends on the value of the controller's input set need not access the entire set of variables of the input set (the *Coeither*) [5].

If the controller's function is y = F(x) and y = (y_1,y_2);

$g = id^{b}_{a}(x)$;

$h = id^{b}_{c}(x)$;

then

SYNTAX: y_1 = A(g) *Coinclude* y_2 = B(h);

SYNTAX: y = A(g) *Coeither* y = B(h);

Structures, in addition to the primitive control structures, e.g., *Cojoin*, can be used to define other structures. For example, the *Whereby*, defined with the *Cojoin*, gives us the facility to use constant symbols as operands of a function.

SYNTAX: *Whereby* y = W(h,CON);

The *Whereby* is used as in

y = x + 1

Here, the constant symbol "1" defines the particular K_{con} operation that makes the instance of the *Whereby* structure a function. Note also that the operator "+" (an instance of W) is used as an infix operator. In AXES we are free to use either prefix or infix notation, as desired.

We can visualize the instance of a structure as being either written down on a piece of paper by a human being, or to a register by a software or hardware process. To check an instance in an HOS system, the use of a *STRUCTURE* is compared with the *STRUCTURE* definition itself by an analyzer. All instances of a structure can be viewed as being supplied to the structure dynamically. The *STRUCTURE* for an asynchronous system, such as an operating system or the Higher Order Machine (HOM) [33], is a recursive relation relating each state of a machine to a previous state of the same machine within an instance of a machine system. To check the instances of an asynchronous machine in a real-time environment, an analyzer is used to check not only the use of the *STRUCTURE* with the *STRUCTURE* definition, itself, also to check to see that all the users of that *STRUCTURE* are consistent.

We indicate the potential happening of each machine instance by specifying a user system to be "*On*" the machine system, e.g., the syntax *Where* A *on* HOM specifies the initial nodal family of system A to be used by the first machine instance and the nodal family for each next recursive instance of the HOM function K_A to be determined by the ordering relationships of the nodal families within system A. A nodal family is a 3-tuple whose members are functions which stand in a particular relation (c.f. Appendix I). By indicating only "HOM" in the syntax of this structure, rather than, for example "y = HOM(x)," the state of the HOM remains hidden from the user.

II. Levels and Layers in AXES

In AXES we emphasize in our notation the separation of the layers within one system or between systems. The layer relationships are defined with the *Where* statement in AXES.

In particular, we distinguish between in *instance of a layer* (representing one performance pass of a system)* and a *layer* (representing all performance passes of a system). We distinguish between *communication within one layer*, which always represents the same instance, and *communication between layers*, which takes place when an instance of one layer communicates with an instance of another layer (e.g., real-time asynchronous processes). We distinguish between (1) the system and the definition of that system, (2) the system and the description of that system, (3) the system and the implementation of that system, and (4) the system and the execution of that system.

A machine is a system which executes another system. There are dedicated machines, asynchronous machines, and asynchronous machines. A dedicated machine always performs the same function. Thus the "mapping" of an AXES *FUNCTION* could be viewed as a dedicated machine. A synchronous machine must only execute one system to completion before another system uses that machine. An AXES *OPERATION* could be viewed as a synchronous machine. An asynchronous machine may execute instances of more than one system before either system reaches execution completion. Thus, an AXES *STRUCTURE* can be viewed as an asynchronous machine.

The environment of a machine must be secure in that (1) a user should not have to be concerned with any of the details that have to do with its execution, and (2) a user should not be allowed to have visibility into another user's environment.

In an asynchronous machine there are several types of data, all of which must be maintained as secure data throughout all instances of the machine. These types of data include (1) temporary values which exist for one or more users, (2) values from another machine with respect to the machine itself as a user, (3) values with respect to the variables of the machine itself, (4) values which are functionally related to a previous instanceof the machine system, (5) values which are functionally related to a previous instance of the machine for a given instance of a user, and (6) values which are functionally related to a previous instance of the machine for a given instance of another machine.

The *definition* (dynamic state) of a system is equivalent to the formal semantics of a system. The *description* (static state) of a system is equivalent to the syntax of a system. The *implementation* (static state which includes a system, a machine to run that system, and the mechanisms necessary to relate that system to the machine) of a system is equivalent to that same system ready to be exercised. The execution (dynamic state which includes a system, a machine running that system, and mechanisms which relate that system to the machine) of a system is that system being exercised by a machine.

Not only must we be aware of the types of system layers, but we also must be aware of how many different definitions, descriptions, implementations, and executions are possible or potentially possible for one system. Most important, we must determine those states which are necessary and those which are not only unnecessary but which are causing serious difficulties in the development of a system. Towards this end it is necessary to have available a means for determining both the types and the nature and number of states within each type [5].

Sometimes the need for the definition of the layers of a system depends on how the system is to be developed and executed. On one project the users might wish to compile source code before a target system is ready to be executed. On another project users might wish to interpret the code in real time. Thus, not only must the layers of a system be determined, but also it must be determined when, how, and where transformations from one layer to another layer take place.

*as opposed to a *level* which is a step of refinement (or more explicit definition) within a given instance of a layer.

In the resource allocation process, a name is assigned to a value or a value to a name. Resource allocation also includes the ability to replace a name by an equivalent name or a value by an equivalent value. Sometimes one layer is produced from another layer by a third layer. Sometimes the description of a layer, as opposed to the layer itself, becomes the object of communication. Sometimes the same layer is resource allocated as a function to one layer and as data to another layer.

The general concepts of layer communication can be related to familiar examples. When two asynchronous processes are in the execution mode, a value from a given process is assigned to a name (or variable) associated with another process. Conversely, that other process assigns a value to the name associated with the first process. When an integer is implemented, a specific representation (or value) for a specific machine is assigned to the name representing the integer. When a compiler compiles an HOL program, it assigns names (or registers) to values in order to translate from one description to another. It also fulfills an implementation function in that operations and data are replaced with specific machine-dependent values. A compiler, in order to operate on an HOL program, must be able to read the description layer of the program as input for the translation process. In addition, it requires further input of its own in order to provide the implementation layer of the program. An OS system has a more complex job in that it is usually required to communicate with both the description of a system and the system itself. When a function is refined to a lower level of more detailed functions, the control integrity of the values and names from the parent layer must be maintained at the layer of the offspring.

Ideally, we want to be able to define a system, as abstractly as possible; describe that system in a syntax we can all relate to; verify that description; implement that system for a machine that will talk directly to that system; and collect the machine mechanisms, and only those mechanisms, that are necessary to execute a given system. In such a way we can eliminate dependencies on a particular primitive machine until the very end of a development process. For when we go from one definition to another, from one description to another, from one implementation to another, or from one execution to another, we are really resource allocating to another machine level. Thus, there should be no need to resource allocate for more than one given layer at a time.

Appendix I and Appendix II References

[1] Hamilton, M. and Zeldin, S. "The Foundations for AXES: A Specification Language Based on Completeness of Control," R-964. The Charles Stark Draper Laboratory, Inc., Cambridge, MA, March 1976.

[2] Hamilton, M. and Zeldin, S. "Higher Order Software--A Methodology for Defining Software." IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976.

[3] Hamilton, M. and Zeldin, S. "Integrated Software Development System/ Higher Order Software Conceptual Description," TR-3, Higher Order Software, Inc., (hereafter cited as HOS, Inc.), Cambridge, MA, Nov. 1976.

[4] Hamilton, M. and Zeldin, S. "AXES Syntax Description," TR-4, HOS, Inc., Cambridge, MA, Dec. 1976.

[5] Hamilton, M. and Zeldin, S. "Operating System Machine with Respect to Resource Allocation" in "Techniques for Operating System Machines," TR-7, HOS, Inc., Cambridge, MA, July 1977.

[6] Hamilton, M. and Zeldin, S. "The Manager as an Abstract Systems Engineer," Digest of Papers, Fall COMPCON 77 (Washington, D.C.), IEEE Computer Society, Cat. No. 77CG1258-3C, Sept. 6-9, 1977.

[7] Searle, J.R. "Review of J.M. Sadock, Toward a Linguistic Theory of Speech Acts," Language 52, 1976.